



# ADVANCED MICROPROCESSORS AND PERIPHERALS

**A K RAY | K M BHURCHANDI**

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.



**Tata McGraw-Hill**

Copyright © 2006, 2000 by Tata McGraw-Hill Publishing Company Limited.

Third reprint 2007

DZLQCRLYRAQVX

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,  
Tata McGraw-Hill Publishing Company Limited.

ISBN 0-07-060658-7

Published by the Tata McGraw-Hill Publishing Company Limited,  
7 West Patel Nagar, New Delhi 110 008, typeset in Times at Script Makers,  
19, A1-B, DDA Market, Pashchim Vihar, New Delhi 110 063 and printed at  
S.P. Printers, E-120, Sector 7, Noida

Cover Printer: SDR Printers

# Contents

<i>Preface to the Second Edition</i>	xiii
<i>Preface to the First Edition</i>	xv
<i>Acknowledgements</i>	xix
<b>I. The Processors: 8086/8088—Architectures, Pin Diagrams and Timing Diagrams</b>	1
1.1 Register Organisation of 8086	2
1.2 Architecture	3
1.3 Signal Descriptions of 8086	8
1.4 Physical Memory Organisation	14
1.5 General Bus Operation	16
1.6 I/O Addressing Capability	17
1.7 Special Processor Activities	18
1.8 Minimum Mode 8086 System and Timings	21
1.9 Maximum Mode 8086 System and Timings	25
1.10 The Processor 8088	28
<i>Summary</i>	36
<i>Exercises</i>	36
<b>2. 8086/8088 Instruction Set and Assembler Directives</b>	38
2.1 Machine Language Instruction Formats	38
2.2 Addressing Modes of 8086	41
2.3 Instruction Set of 8086/8088	46
2.4 Assembler Directives and Operators	74
<i>Summary</i>	82
<i>Exercises</i>	83
<b>3. The Art of Assembly Language Programming with 8086/8088</b>	84
3.1 A Few Machine Level Programs	85
3.2 Machine Coding the Programs	91
3.3 Programming with an Assembler	95
3.4 Assembly Language Example Programs	103
<i>Summary</i>	129
<i>Exercises</i>	129

**4. Special Architectural Features and Related Programming**

131

- 4.1 Introduction to Stack 131**
- 4.2 Stack structure of 8086/88 133**
- 4.3 Interrupts and Interrupt Service Routines 138**
- 4.4 Interrupt Cycle of 8086/8088 138**
- 4.5 Non Maskable Interrupt 141**
- 4.6 Maskable Interrupt (INTR) 141**
- 4.7 Interrupt Programming 142**
- 4.8 Passing Parameters to Procedures 145**
- 4.9 Handling Programs of Size More than 64K 148**
- 4.10 MACROS 150**
- 4.11 Timings and Delays 152**
  - Summary 155*
  - Exercises 155*

**5. Basic Peripherals and Their Interfacing with 8086/88**

157

- 5.1 Semiconductor Memory Interfacing 158**
- 5.2 Dynamic RAM Interfacing 167**
- 5.3 Interfacing I/O Ports 173**
- 5.4 PIO 8255 [Programmable Input-Output Port] 184**
- 5.5 Modes of Operation of 8255 187**
- 5.6 Interfacing Analog to Digital Data Converters 212**
- 5.7 Interfacing Digital to Analog Converters 224**
- 5.8 Stepper Motor Interfacing 228**
- 5.9 Control of High Power Devices Using 8255 231**
  - Summary 232*
  - Exercises 233*

**6. Special Purpose Programmable Peripheral Devices and Their Interfacing**

235

- 6.1 Programmable Interval Timer 8253 235**
- 6.2 Programmable Interrupt Controller 8259A 249**
- 6.3 The Keyboard/Display Controller 8279 266**
- 6.4 Programmable Communication Interface 8251 USART 278**
  - Summary 290*
  - Exercises 290*

**7. DMA, Floppy Disk and CRT Controllers**

293

- 7.1 DMA Controller 8257 294**
- 7.2 DMA Transfers and Operations 300**
- 7.3 Programmable DMA Interface 8237 306**

7.4 Floppy Disk Controller 8272	318
7.5 CRT Controller 8275	350
7.6 CRT Controller 6845	368
<i>Summary</i>	389
<i>Exercises</i>	389
<b>8. Multimicroprocessor Systems</b>	<b>392</b>
8.1 Interconnection Topologies	393
8.2 Software Aspects of Multimicroprocessor Systems	397
8.3 Numeric Processor 8087	399
8.4 I/O Processor 8089	420
8.5 Bus Arbitration and Control	423
8.6 Tightly Coupled and Loosely Coupled Systems	428
8.7 Design of a PC Based Multimicroprocessor System	430
<i>Summary</i>	442
<i>Exercises</i>	442
<b>9. 80286-80287—A Microprocessor with Memory Management and Protection</b>	<b>444</b>
9.1 Salient Features of 80286	444
9.2 Internal Architecture of 80286	446
9.3 Signal Descriptions of 80286	451
9.4 Real Addressing Mode	454
9.5 Protected Virtual Address Mode (PVAM)	455
9.6 Privilege	463
9.7 Protection	468
9.8 Special Operations	470
9.9 80286 Bus Interface	473
9.10 Basic Bus Operations	474
9.11 Fetch Cycles of 80286	475
9.12 80286 Minimum System Configuration	476
9.13 Interfacing Memory and I/O Devices with 80286	478
9.14 Priority of Bus Use by 80286	481
9.15 Bus Hold and HLDA Sequence	484
9.16 Interrupt Acknowledge Sequence	485
9.17 Instruction Set Features	486
9.18 80287 Math Coprocessor	492
<i>Summary</i>	503
<i>Exercises</i>	503
<b>10. 80386-80387 and 80486 the 32-Bit Processors</b>	<b>505</b>
10.1 Salient Features of 80386DX	506
10.2 Architecture and Signal Descriptions of 80386	506

10.3	Register Organization of 80386	510
10.4	Addressing Modes	512
10.5	Data Types of 80386	514
10.6	Real Address Mode of 80386	514
10.7	Protected Mode of 80386	515
10.8	Segmentation	516
10.9	Paging	518
10.10	Virtual 8086 Mode	520
10.11	Enhanced Instruction Set of 80386	523
10.12	The Coprocessor 80387	524
10.13	<u>The CPU with a Numeric Coprocessor—80486DX</u>	<u>528</u>
	<i>Summary</i>	539
	<i>Exercises</i>	540

## **11. Recent Advances in Microprocessor Architectures— A Journey from Pentium Onwards**

**542**

11.1	Salient Features of 80586 (Pentium)	543
11.2	A Few Relevant Concepts of Computer Architecture	544
11.3	System Architecture	544
11.4	Branch Prediction	548
11.5	Enhanced Instruction Set of Pentium	548
11.6	What is MMX?	549
11.7	Intel MMX Architecture	549
11.8	MMX Data Types	550
11.9	Wraparound and Saturation Arithmetic	550
11.10	MMX Instruction Set	550
11.11	Salient Points About Multimedia Application Programming	552
11.12	Journey to Pentium-Pro and Pentium-II	552
11.13	Pentium III (P-III)—The CPU of the Next Millennium	554
	<i>Summary</i>	554
	<i>Exercises</i>	555

## **12. Pentium 4—Processor of the New Millennium**

**556**

12.1	Genesis of Birth of Pentium 4	556
12.2	Salient Features of Pentium 4	557
12.3	Netburst Microarchitecture for Pentium 4	558
12.4	Instruction Translation Lookaside Buffer (ITLB) and Branch Prediction	562
12.5	Why Out of Order Execution	562
12.6	Rapid Execution Module	564
12.7	Memory Subsystem	564
12.8	Hyperthreading Technology	565
12.9	Hyperthreading in Pentium	566

12.10 Extended Instruction Set in Advanced Pentium Processors	568
12.11 Instruction Set Summary	573
<b>12.12 Need for Formal Verification</b>	<b>584</b>
<i>Summary</i>	584
<i>Exercises</i>	584
<b>13. RISC Architecture — An Overview</b>	<b>585</b>
13.1 A Short History of RISC Processors	586
13.2 Hybrid Architecture—RISC and CISC Convergence	586
13.3 The Advantages of RISC	587
13.4 Basic Features of RISC Processors	587
13.5 Design Issues of RISC Processors	588
13.6 Performance Issues in Pipelined Systems	589
13.7 Architecture of Some RISC Processors	591
13.8 Architecture of Some RISC Processors	596
<i>Summary</i>	596
<i>Exercises</i>	596
<b>14. Microprocessor Based Aluminium Smelter Control</b>	<b>598</b>
14.1 General Process Description of an Aluminium Smelter	598
14.2 Normal Control of Electrolysis Cell	599
14.3 Cell Abnormalities in an Aluminium Smelter	600
14.4 Brief Description of the Control Laws for Abnormal Cells	601
14.5 Salient Issues in Design	602
14.6 Smelter Controller Hardware	602
14.7 Control Algorithm	603
<i>Summary</i>	607
<b>15. Design of a Microprocessor Based Pattern Scanner System</b>	<b>608</b>
15.1 Organization of the Scanner System	609
15.2 Description of the Scanning System	611
15.3 Programmed Mode of Operation	613
15.4 Memory Read/Write System and Start-up Procedures	615
15.5 Result and Discussion	616
<i>Summary</i>	618
<b>16. Design of an Electronic Weighing Bridge</b>	<b>619</b>
16.1 Design Issues	620
16.2 Software Development	635
16.3 Calibration	645
<i>Summary</i>	646

<b>17. An Introduction to Microcontrollers 8051 and 80196</b>	<b>647</b>
17.1 Intel's Family of 8-bit Microcontrollers	649
17.2 Architecture of 8051	649
17.3 Signal Descriptions of 8051	652
17.4 Register Set of 8051	654
17.5 Important Operational Features of 8051	655
17.6 Memory and I/O Addressing by 8051	658
17.7 Interrupts of 8051	661
17.8 Instruction Set of 8051	662
17.9 Design of a Microcontroller 8051 Based Length Measurement System for Continuously Rolling Cloth or Paper	665
17.10 Intel's 16-bit Microcontroller Family MCS-96	668
<i>Summary</i>	678
<i>Exercises</i>	679
<i>Appendix A</i>	681
<i>Appendix B</i>	691
<i>Appendix C</i>	707
<i>Index</i>	717

## Preface to the Second Edition

Since the publication of the first edition of this book in 2000, a number of advanced microprocessors, such as Pentium 4, based on Net Burst micro-architecture have arrived on the scene. The recent advancements in the architecture of microprocessors have been quite extensive and the undergraduate and graduate students from diverse engineering as well as science disciplines have become keenly interested in the subject.

The first edition of the book has been extensively used by the students and teachers of diverse disciplines from all over India and also from abroad. We have received a number of valuable suggestions and comments towards the enhancement of the quality of the book. These suggestions along with our personal evaluation of the book primarily formed the basis for the revision of the contents and presentation of this edition.

Two new chapters have been introduced in this edition, while one chapter has been removed. The chapter on 'Pentium 4 – Processor of the new millennium' presents the architectural details of Pentium 4. A set of novel concepts like super scalar execution, execution trace cache, rapid execution engine, streaming SIMD Extension and so on have been presented in this chapter. With the advent of a host of processors, based on hybrid CISC-RISC architecture, it has become imperative for the students to understand the basic features and design issues of RISC processors. Keeping in view of this requirement we have introduced a new chapter 'RISC Architecture – An overview'. The chapter on 'System Software and Operating system' has been deleted from this edition. This chapter may however, be accessed through the Online Learning Centre version of the book. The rest of the chapters have been appropriately modified and recast, keeping in view the needs of the readers.

We hope that like the first edition, this edition of the book will also be immensely appreciated by the readers.

AJOY KUMAR RAY  
KISHOR M BHURCHANDI

# Preface to the First Edition

With the advent of the first 4-bit microprocessor 4004 from Intel Corporation in 1971, there has been a silent revolution in the domain of digital system design, which has shaken many facets of the current technological progress. In the last 28 years the world has seen an evolution of microprocessors, whose impact on today's technological scenario is phenomenal.

This evolution was possible because of the tremendous advances in the semiconductor process technology. The first microprocessor 4004 contained only ten thousand transistors while the component density increased more than threefold in less than a decade's time. Immediately after the introduction of the 4004, Intel introduced the first eight bit microprocessor 8008 in 1972; these processors were, however, not successful because of their inherent limitations. In 1974, Intel released the first general purpose 8-bit microprocessor 8080. This CPU also was not functionally complete and the first 8-bit functionally complete CPU 8085 was introduced in 1977.

The 8085 CPU is still the most popular one amongst all the 8-bit CPUs. The 8085 CPU houses an on-chip clock generator and provides good performance utilizing an optimum set of registers and a reasonably powerful ALU. The major limitations of these 8-bit microprocessors are their limited memory addressing capacity, slow speed of execution, limited number of scratchpad registers and non-availability of complex instruction set and addressing modes. Another important point to be mentioned here is that 8085 does not support adequate pipelining or parallelism which is so important for enhancing the speed of computation. For example, the non-availability of any instruction queue in an 8085 CPU leads to a situation where the fetching of opcode and operands along with the execution takes place in an absolutely sequential manner.

The first 16-bit CPU from Intel was a result of the designers' efforts to produce a more powerful and efficient computing machine. The designers of 8086 CPU had taken note of the major limitations of the previous generations of the 8-bit CPUs. The 8086 contains a set of 16-bit general purpose registers, supports a 16-bit ALU, a rich instruction set and provides segmented memory addressing scheme. The introduction of a set of segment registers for addressing the segmented memory in 8086 was indeed a major step in the process of evolution. All these features made this 16-bit processor a more efficient CPU.

The development of IBM PC started in July 1980, and precisely after one year, the first machine based on Intel 8088 CPU (which is functionally equivalent to 8086 but supports only 8-bit external data bus) with 1 or 2 floppy disk drives, a keyboard and a monochrome monitor was announced in August 1981. The machine operating system was an early version of operating system MS-DOS from Microsoft. In March 1983, a new version of IBM PC called PC-XT was introduced with ten megabyte hard disk, one double side double density floppy disk drive, keyboard, monitor and asynchronous communication adapter. In fact, the introduction of IBM PCs in 1980s had, to a large extent, produced a profound impact on the evolution of microprocessors. With the introduction of each new generation of microprocessors, the performance of the Personal Computers have also been enriched.

The major limitation in 8086 was that it did not have the memory management and protection capabilities, which was considered an extremely important feature deemed to be an integral part of a

CPU of the eighties. 80286 was the first CPU to possess the ability of memory management, privilege and protection. However, the 80286 CPU also had a limitation on the maximum segment size supported by it (only 64 Kb). Another limitation of 80286 was that, once it was switched into protected mode, it was difficult to get it back to real mode. The only way of reverting it to the real mode was to reset the system.

In the mid eighties the more computationally demanding problems necessitated the development of still faster CPUs. Thus appeared 80386 which was the first 32-bit CPU from Intel. The memory management capability of 80286 was enhanced to support virtual memory, paging and four levels of protection. The design of 80386 circumvented this problem. Moreover, the maximum segment size in 80386 was enhanced and this could be as large as 4 Gb with 80386 supporting as many as 16384 segments. The 80386 along with its math coprocessor 80387, provided a high speed environment. 80486 was designed with an integrated math coprocessor. After getting integrated, the speed of execution of mathematical operations enhanced three folds. Also for the first time an 8 Kb four-way set associative code and data cache was introduced in 80486. A five-stage instruction pipelining was also introduced.

The earlier generation CPUs supported rather crude instruction sets. It was not expected that the programmers those days would write large machine code programs. A single high level instruction might be compiled into ten or even hundred machine code operations. In the course of evolution from the early 8-bit CPUs, the trend was to design CPUs, which could support more and more complex instructions at the assembly language level. Designers of complex instruction set computers (CISC) wanted to reduce this gap.

Since the early days of microprocessor development the designers have tried to make them more powerful by designing more complex instructions. But then some of these powerful instructions and addressing modes were hardly used by the programmers. In fact some of these instructions' logic took up a large part of the microprocessors' silicon chip. The reduced instruction set computer (RISC) designers observed that the data movement type of machine instructions are frequently executed by the CPU. They have optimised the CPUs to execute these instructions rapidly. RISC provided a regular set of instructions having the same format with a lot of pipelining. To improve the processor's performance, the possible ways are suggested below.

- (a) Increasing the processor and system clock rate.
- (b) Optimising and improving the instruction set.
- (c) Executing multiple instructions in one cycle and incorporating parallelism in the CPU architecture.

The first option is applicable both to CISC and RISC processors. The second option is primarily for CISC but is applicable to RISC as well. The third option is more suited to RISC CPUs. Ever since the appearance of commercially available RISC CPUs, there has been a debate over the performance of RISC versus CISC. The RISC architects argue that their instructions may be executed in a single cycle and thus take less time than is taken by a CISC CPU. This is because of pipelining, reduction of instructions to a simple operation and synthesis of complex operations with compiler generated code sequences. When RISC machines first arrived in the market, CISC processors were performing at 6–10 cycles per instruction, while the RISC CPUs could execute a set of simpler instructions in one cycle and offer better performance. Many of the CISC processors have subsequently used many features of RISC.

This book is intended as a textbook on 'Advanced Microprocessors' which is a compulsory course at graduate and postgraduate levels in many science and engineering branches of studies, specially in

Electronics, Electrical, Instrumentation, Physics and Computer Science disciplines. The book is suitable for a one-semester course on advanced microprocessor—their architectures, programming, hardware interfacing and applications. The purpose of our book is to provide the readers with a good foundation on microprocessors, their principles and practices. We have tried to keep an appropriate balance between the basic concepts and practical applications related to microprocessors technology. Thus we have aimed at the following:

- To present the fundamental concepts of advanced microprocessors and their architectures.
- To enable the students to write efficient programs in assembly level language of the 8086 family of microprocessors.
- To make the students aware of the techniques of interfacing between the processors and peripheral devices so that they themselves can design and develop a complete microprocessor based system.
- To present in a lucid manner the basic concepts of systems programming, viz., operating systems, assemblers, compilers, etc. to enable the students to understand the entire space of microprocessor technology and specially the software aspects related to microprocessing.
- To present to the students the utility of faster modes of data transfer and techniques.
- To present a host of interesting applications involving microprocessors.

Some of the salient features of the book are listed below.

1. The book covers a wide range of microprocessors from 16-bit 8086 to Pentium in a lucid manner. The evolution from one processor architecture to another is evident as one goes through the chapters. A detailed description of each microprocessor has been presented in individual Chapters. Chapter 1 covers 8086/8088 architecture in adequate detail. Chapter 9 covers 80286 along with its coprocessor. Chapter 10 covers the microprocessor 80386 and its coprocessor 80387. This chapter also covers in sufficient detail 80486, the integrated CPU with built-in math coprocessor. Pentium, the latest in the Intel microprocessor family, has been briefly presented in Chapter 11.
2. An important feature of the book is the inclusion of a number of interesting applications of microprocessors. An adequate account of each one of these applications has been presented in the book. An interesting application of microprocessors for controlling an Aluminium Smelter has been presented in Chapter 13. Chapter 14 presents another interesting application in the area of Pattern Scanner Design. Design of a microprocessor-based Electronic Weighing Bridge has been elaborated in Chapter 15.
3. One of the major problems encountered by students is the difficulty in writing assembly language programs. In this book, a large number of assembly language programs have been presented. Which will enable the students to write efficient codes on 16-bit or 32-bit platforms. Chapter 2 covers the 8086 family instruction set and the assembler directives with necessary examples. The art of programming in 8086 Assembly language has been elaborated with a large number of program examples in Chapter 3. A very important spectrum of programs involving stacks, subroutines, interrupts, macros and time delays has been discussed in adequate detail in Chapter 4.
4. A good account of a number of general peripheral devices like I/O ports, keyboards, displays, ADCs, DACs, stepper motors, etc. has been elaborated in Chapter 5.

5. Some special dedicated peripherals like interrupt controllers, DMA controllers, CRT controllers, floppy disk controllers, etc. have been discussed elaborately along with interfacing examples and programs in Chapters 6 and 7. The detailed knowledge about these peripherals is extremely important for interfacing these devices with advanced CPUs and also for designing standalone microprocessor-based systems.
6. Usually the students look for separate books for understanding the system programs' concepts. In this book a complete chapter (Chapter 12) has been devoted to explaining the concepts of assemblers, loaders, linkers, compilers and the operating systems. The understanding of systems programming concepts are extremely important for an integrated understanding of a microcomputer system.
7. The importance of multiprocessor based system design cannot be underestimated in today's world. A full chapter has been devoted to presenting issues related to the multiprocessor based system design. The co-processor like 8087, 8089, etc. along with their interfacing strategies have been presented in Chapter 8 of the book. Design of an 8088 based multimicroprocessor system has been described in adequate detail in this chapter with an example.
8. Microcontrollers are being extensively used in today's industrial environments. An introductory chapter on microcontrollers has been included for the benefit of the students. Intel's 8-bit microcontroller 8051 and 16-bit microcontroller 80196 have been presented in sufficient detail in Chapter 16 along with an 8031 based application design example.  
On the whole, the book is intended to provide adequate support to the reader for acquiring an integrated understanding of the subject.

AJOY KUMAR RAY  
KISHOR M BHURCHANDI

# Acknowledgements

A number of individuals have contributed to the preparation of the manuscript of this book. The authors gratefully acknowledge the contributions of each one of them. The authors would like to express their thanks to the faculty members of IIT Kharagpur and Shri Ramdeobaba Kamla Nehru Engineering College, Nagpur. The authors convey their gratitude to the large number of teachers and students belonging to many science and engineering institutions from diverse parts of India and abroad for their keen interest in this book, which has prompted the authors to undertake the publication of the second edition of the book.

In particular, Prof. G S Sanyal, former Director and Advisor, IIT Kharagpur, Prof. A K Majumdar, Head, Computer Science and Engineering Department IIT Kharagpur, Dr. Tinku Acharya, CTO, Avisere Inc., USA have always shown their active interest in the publication of the book. We are thankful to them. The authors are thankful to Intel Corporation, USA for allowing them to use some of their product specifications for the book.

Some faculty members and students of IIT Kharagpur have contributed in various ways for the completion of this work. The authors would like to acknowledge the services rendered by them. In particular we thank Rajat Sethi of Computer Science and Engineering, who has contributed in shaping the chapter on Pentium 4. Anoop C V, Rahul Gupta, U V Srinadh, Maj. Nilesh Ingle, Maj. Vivek Vaidyanathan of Electronics & ECE department, IIT Kharagpur have reviewed some of the chapters in the second edition and we thank them all.

We thank Prof. D Sarkar, A K Ghosh, Ananda Mitra and. S S Biswas for providing valuable inputs for the chapter on Microprocessor Based Aluminium Smelter Design, Prof B P Sandilya, Tamalika Chakraborty, D S Deshpande, Amit Chatterji, S Verma, S Sukumar, Mainak Chowdhary, Rajat Subhra Mukherji, Arnab Chakraborty, Abha Jain, Smarajana Mishra, Animesh Khemka and S Dihidar deserve special mention for going through parts of the manuscript. We acknowledge the help rendered by Kaushik Mallick Arumoy Mukherjee, Arindam Mandal, Rana Pratap Ghoshal and many others for their help in preparation of the manuscript.

The authors are thankful to the Principal Dr S S Limaye and Management of Shri Ramdeobaba Kamla Nehru Engineering College, Nagpur for their appreciation and encouragement.

We also thank Dr P M Navghare, former Professor VNIT, Nagpur and Professor, Department of Engineering, University of Eritrea, South Africa for his review and valuable suggestions about organization and scope of this book.

Our thanks are also due to Prof. R V Kshirsagar, Chairman Electronics Engineering Board of Studies, Nagpur University and Prof. D P Dave, Head IT Dept, Tolani Maritime Institute, Pune for their suggestions towards finalization of this text. We also thank our colleagues Professors P T Karule, A S Khobragade, A G Kothari, A V Gokhale, P R Rothe, S Y Ambatkar and P M More for sharing their concepts and ideas during the preparation of the manuscript. We are grateful to Professors K K Bhoyar, S J Karale, Ms Padma Rao for compilation of the chapter on System Software and Operating Systems which is now going to be on the web site of the book.

We would like to express our thanks and appreciation to Professors R S Pande and Ms R R Harkare who have shown keen interest and encouragement for the publication of the second edition. We take this opportunity to express our gratitude to Prof M. Wasim Khanoomi and Prof V Nitnaware for meticulously reading the proofs for second edition and also for preparation of solution manual of this book. We are also thankful to Prof Anil Srirao, Prof Nitin Narkhede, Prof Ms R S Asamwar, Prof Ms Rajshree Raut, Prof Ms P K Parlewar in Department of Electronics and Communication Engineering of Ramdeobaba Kamla Nehru Engineering College, Nagpur for their critical evaluation of the book. Our thanks are due to Professor P A Dwaramwar, R Khobragade, Rajesh Raut and V E Khetade, for their support during preparation of the manuscript.

Our heartfelt thanks are due to Manish Hote for typing and compiling considerable part of this text and Ms. Namita Gupta of Computer Complex, Nagpur for the dedicated and sincere efforts in the enormous DTP work.

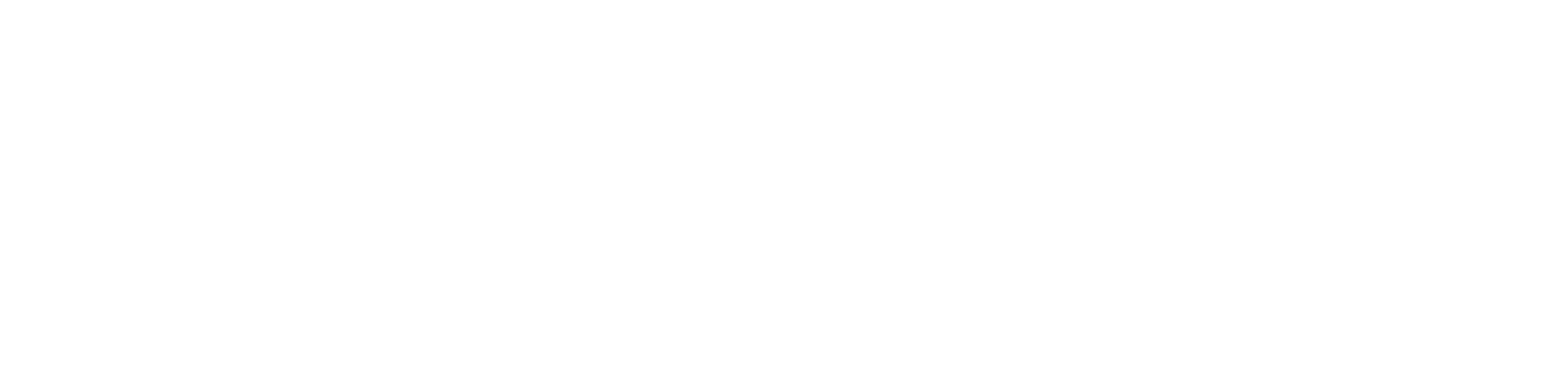
AJOY KUMAR RAY  
KISHOR M BHURCHANDI

# The Processors: 8086/8088—Architectures, Pin Diagrams and Timing Diagrams

## INTRODUCTION

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

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

### 1.1.2 Segment Registers

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

### 1.1.3 Pointers and Index Registers

The pointers contain offset within the particular segments. The pointers IP, BP and SP usually contain offsets within the code (JP), and stack (BP & SP) segments. The *index registers* are used as general purpose registers as well as for offset storage in case of indexed, based indexed and relative based indexed addressing modes. The register SI is generally used to store the offset of source data in data segment while the register DI is used to store the offset of destination in data or extra segment. The index registers are particularly useful for string manipulations.

### 1.1.4 Flag Register

The 8086 *flag register* contents indicate the results of computations in the ALU. It also contains some flag bits to control the CPU operations. Details of the flag register are discussed later in this chapter.

## 1.2 ARCHITECTURE

The architecture of 8086 provides a number of improvements over 8085 architecture. It supports a 16-bit ALU, a set of 16-bit registers and provides segmented memory addressing capability, a rich instruction set, powerful interrupt structure, fetched instruction queue for overlapped fetching and execution etc. The internal block diagram, shown in Fig.1.2, describes the overall organization of different units inside the chip.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

a *predecoding instruction byte queue* (6 bytes long). The bus interface unit makes the system's bus signals available for external interfacing of the devices. In other words, this unit is responsible for establishing communications with external devices and peripherals including memory via the bus. As already stated, the 8086 addresses a segmented memory. The complete physical address which is 20-bits long is generated using segment and offset registers, each 16-bits long.

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

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

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

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

The execution unit contains the register set of 8086 except segment registers and IP. It has a 16-bit ALU, able to perform arithmetic and logic operations. The 16-bit flag register reflects the results of execution by the ALU. The decoding unit decodes the opcode bytes issued from the instruction byte queue. The timing and control unit derives the necessary control signals to execute the instruction



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

		8086	Maximum mode	Minimum mode
GND	1		40	VCC
AD <sub>14</sub>	2		39	AD <sub>15</sub>
AD <sub>13</sub>	3		38	A <sub>16/S<sub>3</sub></sub>
AD <sub>12</sub>	4		37	A <sub>17/S<sub>4</sub></sub>
AD <sub>11</sub>	5		36	A <sub>18/S<sub>5</sub></sub>
AD <sub>10</sub>	6		35	A <sub>19/S<sub>6</sub></sub>
AD <sub>9</sub>	7		34	BHE/S <sub>7</sub>
AD <sub>8</sub>	8		33	MN/MX
AD <sub>7</sub>	9		32	RD
AD <sub>6</sub>	10		31	RQ/GT <sub>0</sub> (HOLD)
AD <sub>5</sub>	11		30	RQ/GT <sub>1</sub> (HLDA)
AD <sub>4</sub>	12		29	LOCK (WR)
AD <sub>3</sub>	13		28	S <sub>2</sub> (M/I <sub>O</sub> )
AD <sub>2</sub>	14		27	S <sub>1</sub> (DT/R)
AD <sub>1</sub>	15		26	S <sub>0</sub> (DEN)
AD <sub>0</sub>	16		25	QS <sub>0</sub> (ALE)
NMI	17		24	QS <sub>1</sub> (INTA)
INTR	18		23	TEST
CLK	19		22	READY
GND	20		21	RESET

**Fig. 1.5 Pin Configuration of 8086**

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

**AD<sub>15</sub>-AD<sub>0</sub>** These are the time multiplexed memory I/O address and data lines. Address remains on the lines during T<sub>1</sub> state, while the data is available on the data bus during T<sub>2</sub>, T<sub>3</sub>, T<sub>w</sub> and T<sub>4</sub>. Here T<sub>1</sub>, T<sub>2</sub>, T<sub>3</sub>, T<sub>4</sub> and T<sub>w</sub> are the clock states of a machine cycle. T<sub>w</sub> is a wait state. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.

**A<sub>19/S<sub>6</sub></sub>, A<sub>18/S<sub>5</sub></sub>, A<sub>17/S<sub>4</sub></sub>, A<sub>16/S<sub>3</sub></sub>** These are the time multiplexed address and status lines. During T<sub>1</sub>, these are the most significant address lines for memory operations. During I/O operations, these lines are low. During memory or I/O operations, status information is available on those lines for T<sub>2</sub>, T<sub>3</sub>, T<sub>w</sub> and T<sub>4</sub>. The status of the interrupt enable flag bit (displayed on S<sub>5</sub>) is updated at the beginning of each clock cycle. The S<sub>4</sub> and S<sub>3</sub> together indicate which segment register is presently being used for memory accesses, as shown in Table 1.1. These lines float to tri-state off (tristated) during the local bus hold acknowledge. The status line S<sub>6</sub> is always low (logical). The address bits are separated from the status bits using latches controlled by the ALE signal.

**Table 1.1 Bus High Enable/status**

<i>S<sub>4</sub></i>	<i>S<sub>3</sub></i>	<i>Indications</i>
0	0	Alternate Data
0	1	Stack
1	0	Code or none
1	1	Data

**BHE/S<sub>7</sub>-Bus High Enable/Status** The bus high enable signal is used to indicate the transfer of data over the higher order (D<sub>15</sub>-D<sub>8</sub>) data bus as shown in Table 1.2. It goes low for the data transfers over D<sub>15</sub>-D<sub>8</sub> and is used to derive chip selects of odd address memory bank or peripherals. BHE is low during T<sub>1</sub> for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on the higher byte of the data bus. The status information is available during T<sub>2</sub>, T<sub>3</sub> and T<sub>4</sub>. The signal is active low and is tristated during 'hold'. It is low during T<sub>1</sub> for the first pulse of the interrupt acknowledge cycle. S<sub>7</sub> is not currently used.

**Table 1.2**

<i>BHE</i>	<i>A<sub>0</sub></i>	<i>Indication</i>
0	0	Whole word
0	1	Upper byte from or to odd address.
1	0	Lower byte from or to even address
1	1	None

**RD -Read** Read signal, when low, indicates the peripherals that the processor is performing a memory or I/O read operation. RD is active low and shows the state for T<sub>2</sub>, T<sub>3</sub>, T<sub>w</sub> of any read cycle. The signal remains tristated during the 'hold acknowledge'.

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

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

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

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

**RESET**: This input causes the processor to terminate the current activity and start execution from FFFF0H. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronised.

**CLK-Clock Input**: The clock input provides the basic timing for processor operation and bus control activity. It's an asymmetric square wave with 33% duty cycle. The range of frequency for different 8086 versions is from 5MHz to 10MHz.

**V<sub>CC</sub>**: +5V power supply for the operation of the internal circuit.

**GND**: ground for the internal circuit.

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

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

**M/I/O -Memory/IO**: This is a status line logically equivalent to  $\bar{S}_2$  in the maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active in the previous  $T_4$  and remains active till final  $T_4$  of the current cycle. It is tristated during local bus "hold acknowledge".

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

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

**DT/R -Data Transmit/Receive**: This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low. Logically, this is equivalent to  $\bar{S}_1$  in maximum mode. Its timing is the same as M/I/O . This is tristated during 'hold acknowledge'.

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

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

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

1. The request occurs on or before  $T_2$  state of the current cycle
2. The current cycle is not operating over the lower byte of a word (or operating on an odd address)

3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence
4. A Lock instruction is not being executed.

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

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

**S<sub>2</sub>, S<sub>1</sub>, S<sub>0</sub>-Status Lines** These are the status lines which indicate the type of operation, being carried out by the processor. These become active during T<sub>4</sub> of the previous cycle and remain active during T<sub>1</sub> and T<sub>2</sub> of the current bus cycle. The status lines return to passive state during T<sub>3</sub> of the current bus cycle so that they may again become active for the next bus cycle during T<sub>4</sub>. Any change in these lines during T<sub>3</sub> indicates the starting of a new cycle, and return to passive state indicates end of the bus cycle. These status lines are encoded in Table 1.3.

**Table 1.3**

S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	Indication
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive

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

**QS<sub>1</sub>, QS<sub>0</sub>-Queue Status** These lines give information about the status of the code-prefetch queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as shown in Table 1.4.

**Table 1.4**

QS <sub>1</sub>	QS <sub>0</sub>	Indication
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty queue
1	1	Subsequent byte from the queue

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

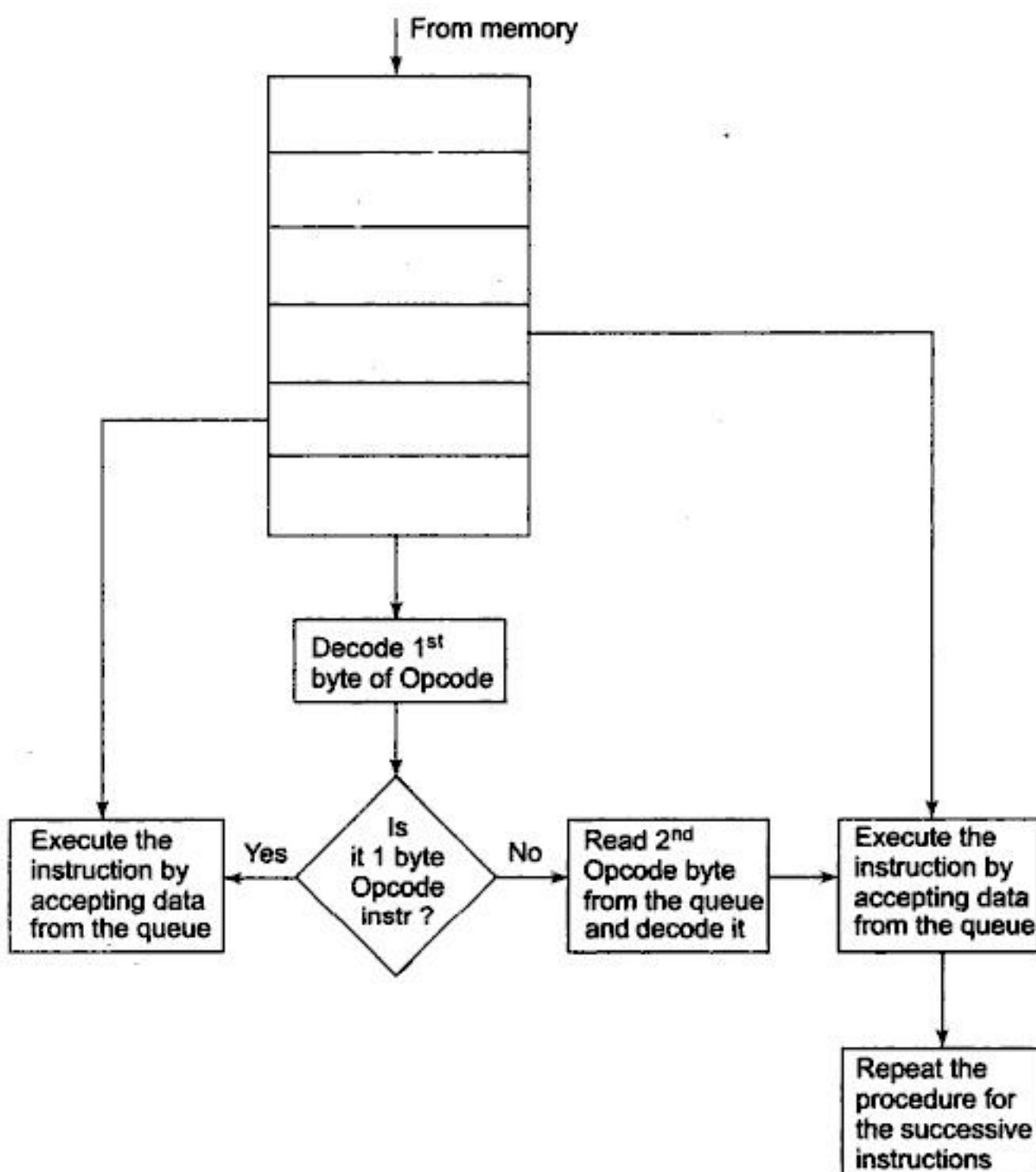
In the beginning, the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS:IP address is odd or two bytes at a time, if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

The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The main point to be noted here is, that the fetch operation of the next instruction is overlapped with the execution of the current instruction. As shown in the architecture, there are two separate units, namely, the execution unit and the bus interface unit. While the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status. Figure 1.6 explains the queue operation.

**RQ/GT<sub>0</sub>, RQ/GT<sub>1</sub>-Request/Grant** These pins are used by other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor's current bus cycle.

Each of the pins is bidirectional with RQ/GT<sub>0</sub> having higher priority than RQ/GT<sub>1</sub>. RQ/GT pins have internal pull-up resistors and may be left unconnected. The request/grant sequence is as follows:

1. A pulse one clock wide from another bus master requests the bus access to 8086.
2. During T<sub>4</sub> (current) or T<sub>1</sub> (next) clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state in the next clock cycle. The CPU's bus interface unit is likely to be disconnected from the local bus of the system.
3. A one clock wide pulse from the another master indicates to 8086 that the 'hold' request is about to end and the 8086 may regain control of the local bus at the next clock cycle.



**Fig. 1.6 The Queue Operation**

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

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

#### 1.4 PHYSICAL MEMORY ORGANISATION

In an 8086 based system, the 1Mbytes memory is physically organised as an odd bank and an even bank, each of 512 Kbytes, addressed in parallel by the processor. Byte data with an even address is transferred on  $D_7 - D_0$ , while the byte data with an odd address is transferred on  $D_{15} - D_8$  buss lines. The processor provides two enable signals, BHE and  $A_0$  for selection of either even or odd or both the

banks. The instruction stream is fetched from memory as words and is addressed internally by the processor as necessary. In other words, if the processor fetches a word (consecutive two bytes) from memory, there are different possibilities, like:

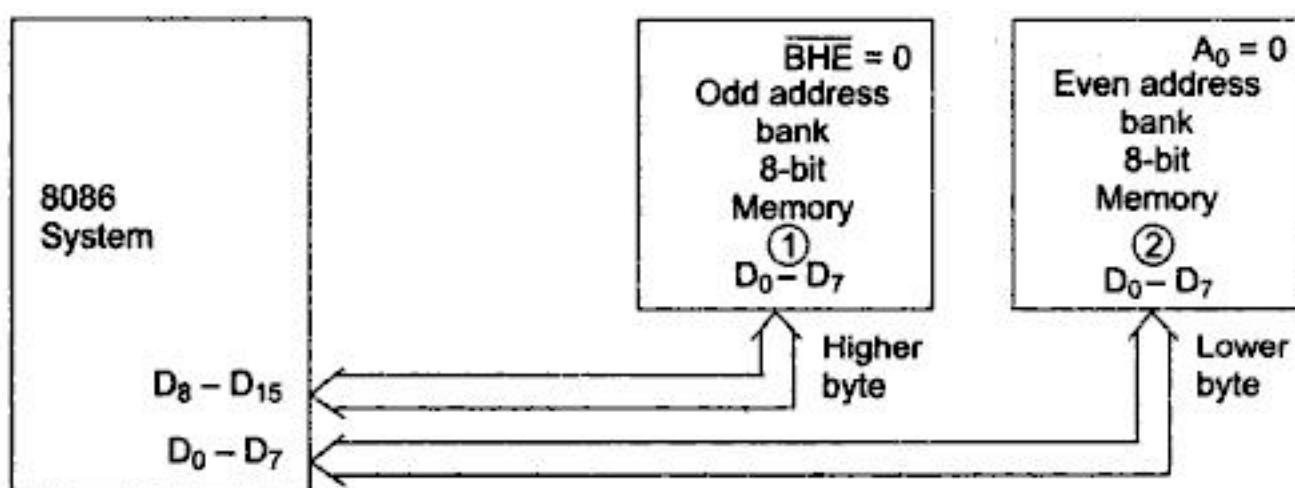
1. Both the bytes may be data operands
2. Both the bytes may contain opcode bits
3. One of the bytes may be opcode while the other may be data

All the above possibilities are taken care of by the internal decoder circuit of the microprocessor. The opcodes and operands are identified by the internal decoder circuit which further derives the signals those act as input to the timing and control unit. The timing and control unit then derives all the signals required for execution of the instruction.

While referring to word data, the BIU requires one or two memory cycles, depending upon whether the starting byte is located at an even or odd address. It is always better to locate the word data at an even address. To read or write a complete word from/to memory, if it is located at an even address, only one read or write cycle is required. If the word is located at an odd address, the first read or write cycle is required for accessing the lower byte while the second one is required for accessing the upper byte. Thus, two bus cycles are required, if a word is located at an odd address. It should be kept in mind that while initialising the structures like stack they should be initialised at an even address for efficient operation.

8086 is a 16-bit microprocessor and hence can access two bytes of data in one memory or I/O read or write operation. But the commercially available memory chips are only byte size, i.e. they can store only one byte in a memory location. Obviously, to store 16-bit data, two successive memory locations are used and the lower byte of 16-bit data can be stored in the first memory location while the second byte is stored in the next location. In a sixteen bit read or write operation both of these bytes will be read or written in a single machine cycle.

A map of an 8086 memory system starts at 00000H and ends at FFFFFH. 8086 being a 16-bit processor is expected to access 16-bit data to / from 8-bit commercially available memory chips in parallel, as shown below in Fig. 1.7.



**Fig. 1.7 Physical Memory Organization**

Thus, bits  $D_0 - D_7$  of a 16-bit data will be transferred over  $D_0 - D_7$  (lower byte) of 16-bit data bus to / from 8-bit memory (2) and bit  $D_8 - D_{15}$  of the 16-bit data will be transferred over  $D_8 - D_{15}$  (higher byte) of the 16-bit data bus of the microprocessor, to / from 8-bit memory (1). Thus to achieve 16-bit data transfer using 8-bit memories, in parallel, the map of the complete system byte memory addresses will obviously be divided into the two memory banks as shown in Fig. 1.7.

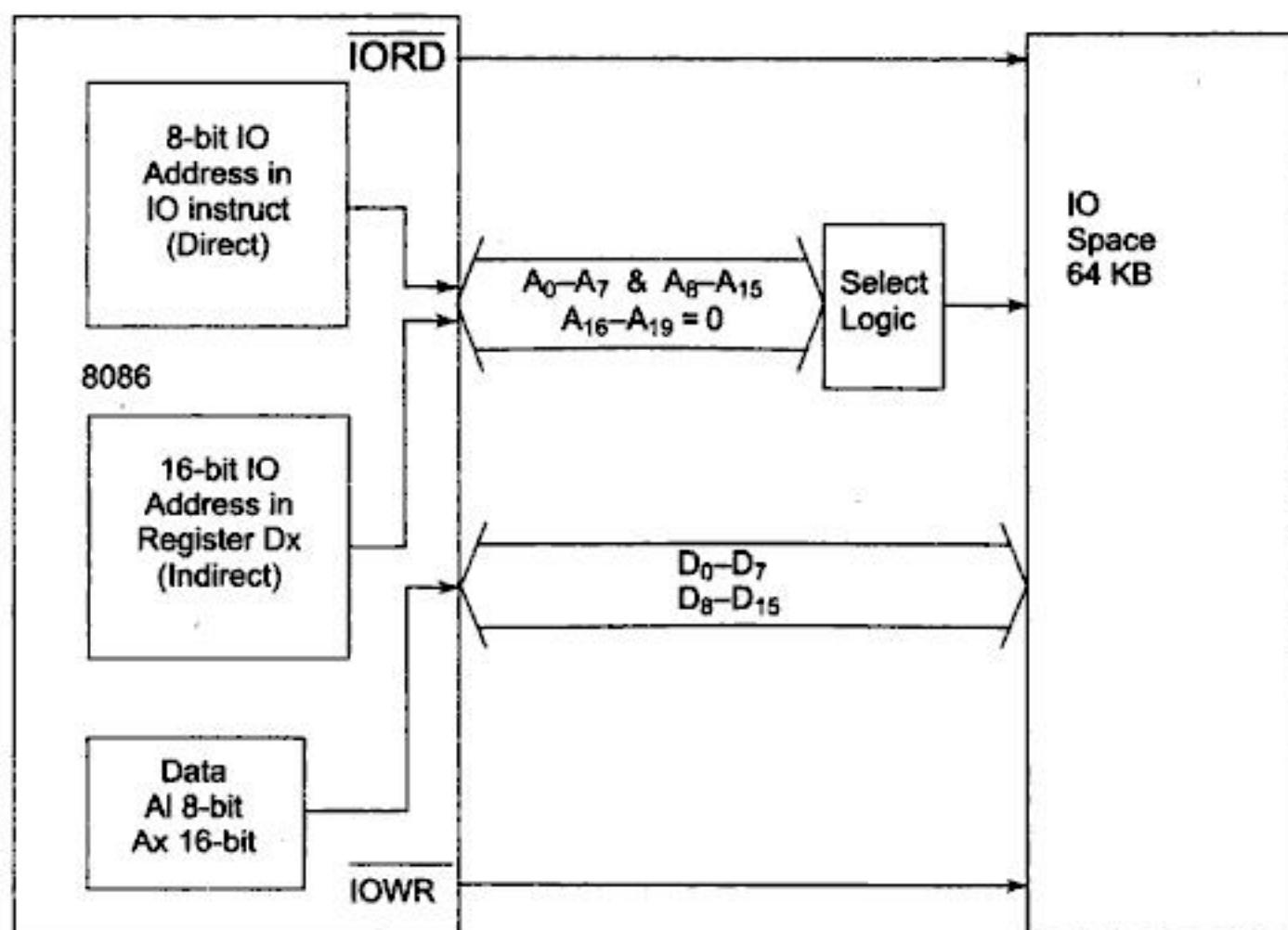


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The 16-bit register DX is used as 16-bit I/O address pointer, with full capability to address up to 64K devices. In this case, the I/O ports are addressed in the same manner as memory locations in the based addressing mode using BX. In memory mapped I/O interfacing, the I/O device addresses are treated as memory locations in page 0, i.e. segment address 0000H. Even addressed bytes are transferred on D<sub>7</sub>-D<sub>0</sub> and odd addressed bytes are transferred on D<sub>8</sub>-D<sub>15</sub> lines. While designing any 8-bit I/O system around 8086, care must be taken that all the byte registers in the system should be even addressed. Figure 1.9 shows 8086 IO addressing scheme.



**Fig. 1.9 8086 IO Addressing**

## 1.7 SPECIAL PROCESSOR ACTIVITIES

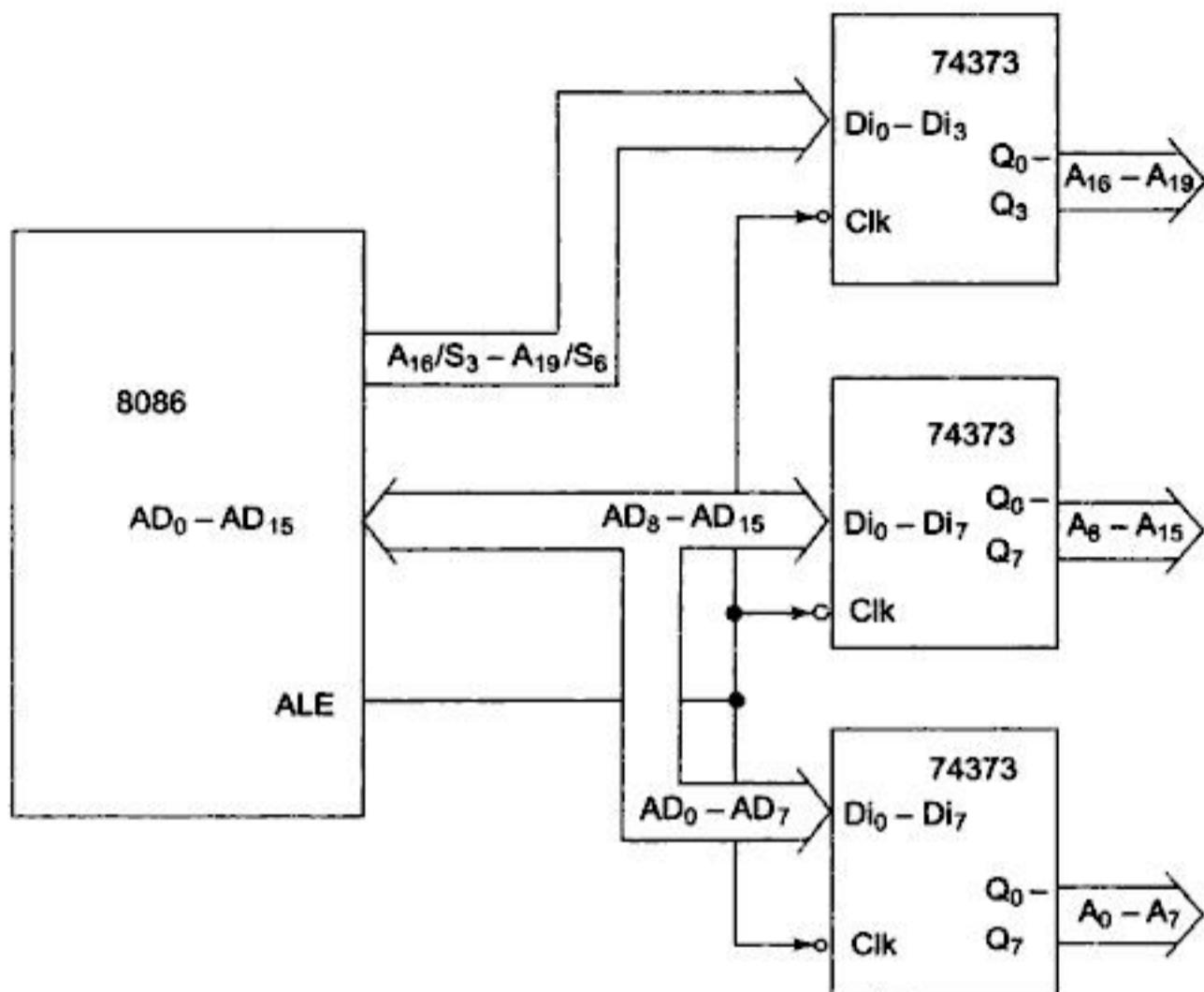
### 1.7.1 Processor Reset and Initialisation

When logic 1 is applied to the RESET pin of the microprocessor, it is reset. It remains in this state till logic 0 is again applied to the RESET pin. The 8086 terminates the on-going operation on the positive edge of the reset signal. When the negative edge is detected, the reset sequence starts and is continued for nearly 10 clock cycles. During this period, all the internal register contents are set to 0000H except CS is set to value FFFFH . Thus, the execution starts again from the physical address FFFF0H. Due to this, the EPROM in an 8086 system is interfaced so as to have the physical memory locations FFFF0H to FFFFFH in it, i.e. at the end of the map.

For the reset signal to be accepted by 8086, it must be high for at least 4 clock cycles. From the instant the power is on, the reset pulse should not be applied to 8086 before 50  $\mu$ s to allow proper initialisation of 8086. In the reset state, all 3-state outputs are tristated. Status signals are active in idle



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Fig.1.10 Latching 20-Bit Address of 8086**

8086 has multiplexed 16-bit data bus in the form of  $AD_0 - AD_{15}$ . The data can be separated from the address and buffered using two bidirectional buffers 74245. It may be noted that the data can either be transferred from microprocessor to memory or from memory to microprocessor in case of write or read operations respectively hence bidirectional buffers are required for deriving the data bus. The signals

$\overline{DEN}$  and  $DT / \overline{R}$  indicate the presence of data on the bus and the direction of the data, i.e. to / from the microprocessor. They are used to drive the chip select (enable) and direction pins of the buffers as indicated below in Fig. 1.11.

If  $\overline{DEN}$  is low it indicates that the data is available on the multiplexed bus and both the buffers (74245) are enabled to transfer data. When DIR pin goes high the data available at X pins of 74245 are transferred to Y pins, i.e. data is transmitted from microprocessor to either memory or IO device (write operation). If DIR pin goes low the data available at Y pins of 74245 is transferred to X pins, i.e. data is received by microprocessor from memory or IO device (read operation).

For deriving control bus from the available control signals  $\overline{RD}$ ,  $\overline{WR}$  and  $M / \overline{IO}$  in case of minimum mode of operation any combinational logic circuit may be used as shown in Fig. 1.12 (a) and Fig. 1.12 (b).

In case of maximum mode of operation a chip bus controller derives all the control signals using status signals  $\overline{S}_0, \overline{S}_1$  and  $\overline{S}_2$ .



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The latches are generally buffered output D-type flip-flops, like, 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086. Transreceivers are the bidirectional buffers and sometimes they are called data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal. They are controlled by two signals, namely,  $\overline{DEN}$  and  $DT/R$ . The  $\overline{DEN}$  signal indicates that the valid data is available on the data bus, while  $DT/R$  indicates the direction of data, i.e. from / to the processor. The system contains memory for the monitor and users program storage. Usually, EPROMS are used for monitor storage, while RAMs for users' program storage. A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices. The clock generator (IC8284) generates the clock from the crystal oscillator and then shapes it to make it more precise so that it can be used as an accurate timing reference for the system. The clock generator also synchronizes some external signals with the system clock. The general system organisation is shown in Fig. 1.13. Since it has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation.

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

The read cycle begins in  $T_1$  with the assertion of the Address Latch Enable (ALE) signal and  $M/\overline{IO}$  signal. During the negative going edge of this signal, the valid address is latched on the local bus. The  $\overline{BHE}$  and  $A_0$  signals address low, high or both bytes. From  $T_1$  to  $T_4$ , the  $M/\overline{IO}$  signal indicates a memory or I/O operation. At  $T_2$ , the address is removed from the local bus and is sent to the output. The bus is then tristated. The Read ( $\overline{RD}$ ) control signal is also activated in  $T_2$ . This signal causes the addressed device to enable its data bus drivers. After  $\overline{RD}$  goes low, the valid data is available on the data bus. The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers. CS logic indicates chip select logic and 'e' and 'O' suffixes indicate even and odd address memory banks.

A write cycle also begins with the assertion of ALE and the emission of the address. The  $M/\overline{IO}$  signal is again asserted to indicate a memory or I/O operation. In  $T_2$ , after sending the address in  $T_1$ , the processor sends the data to be written to the addressed location. The data remains on the bus until the middle of  $T_4$  state. The  $\overline{WR}$  becomes active at the beginning of  $T_2$  (unlike  $\overline{RD}$  is somewhat delayed in  $T_2$  to provide time for floating).

The  $\overline{BHE}$  and  $A_0$  signals are used to select the proper byte or bytes of memory or I/O word to be read or written as already discussed in the signal description section of this chapter.

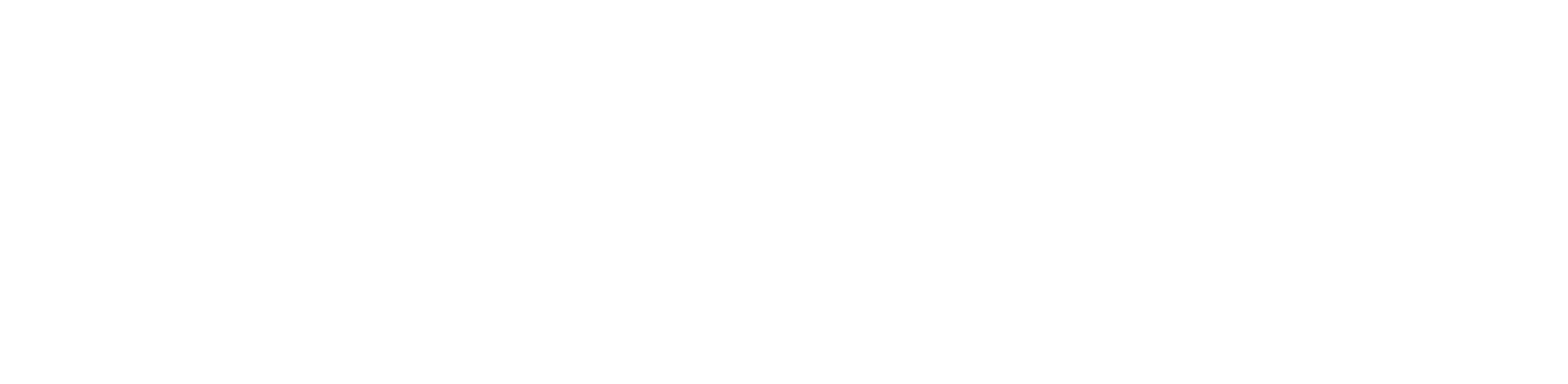
The  $M/\overline{IO}$ ,  $\overline{RD}$  and  $\overline{WR}$  signals indicate the types of data transfer as specified in Table 1.5.

**Table 1.5**

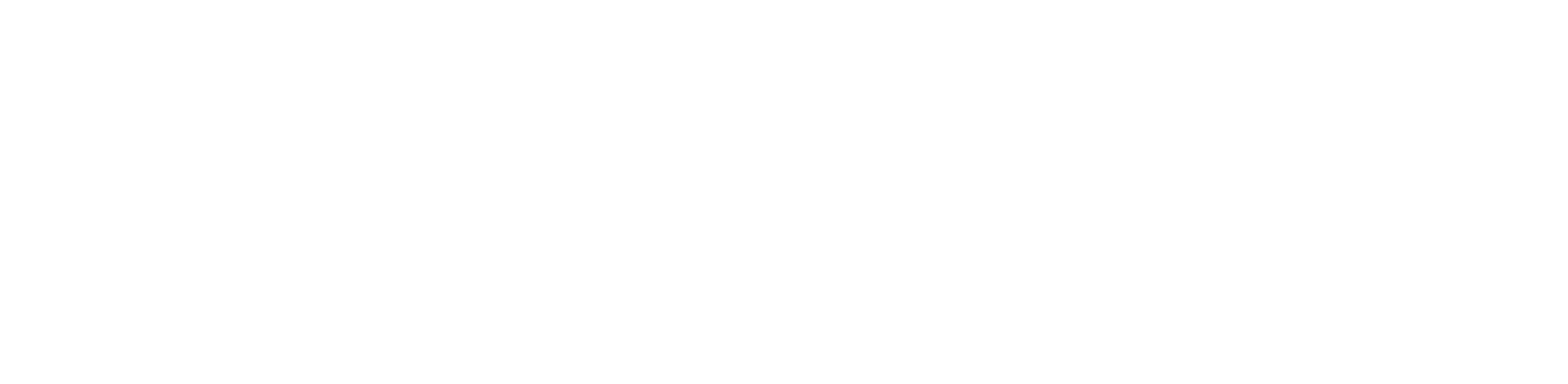
$M/\overline{IO}$	$\overline{RD}$	$\overline{DEN}$	Transfer Type
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



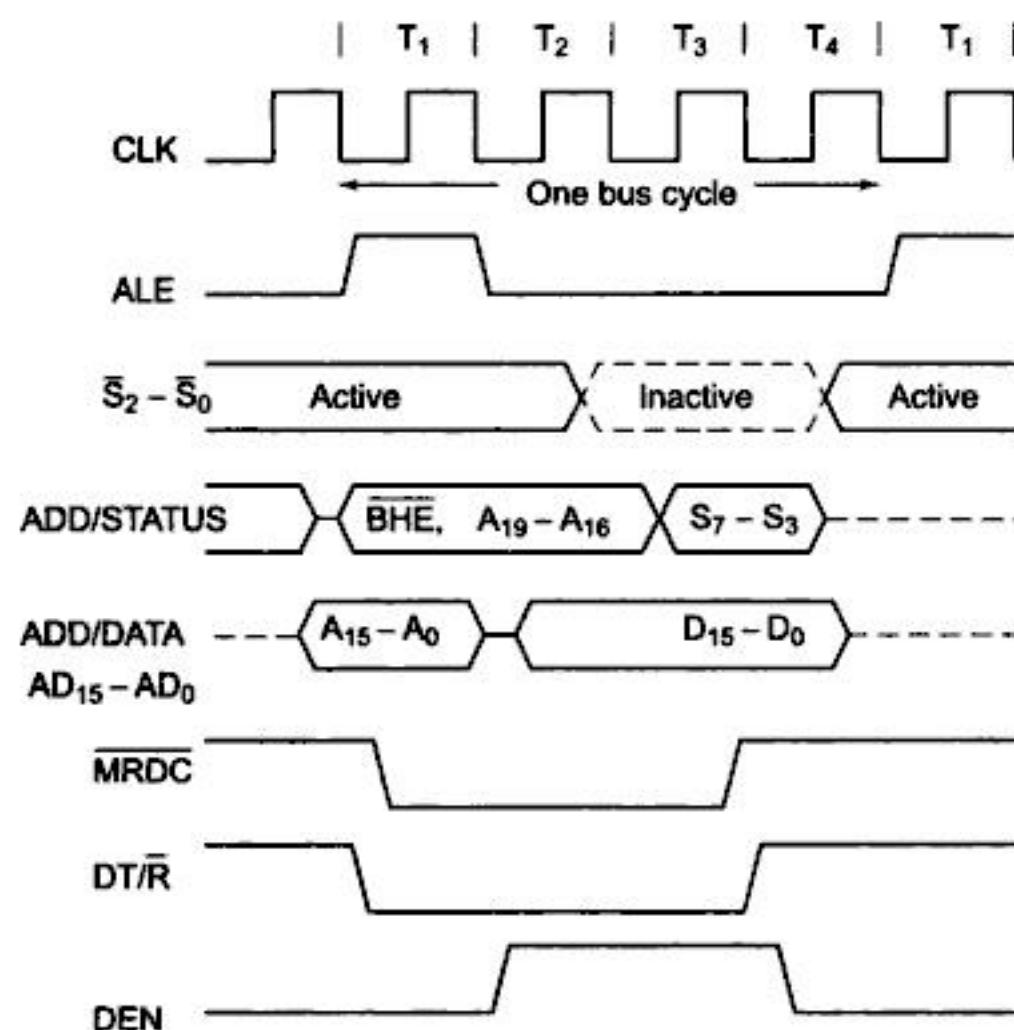
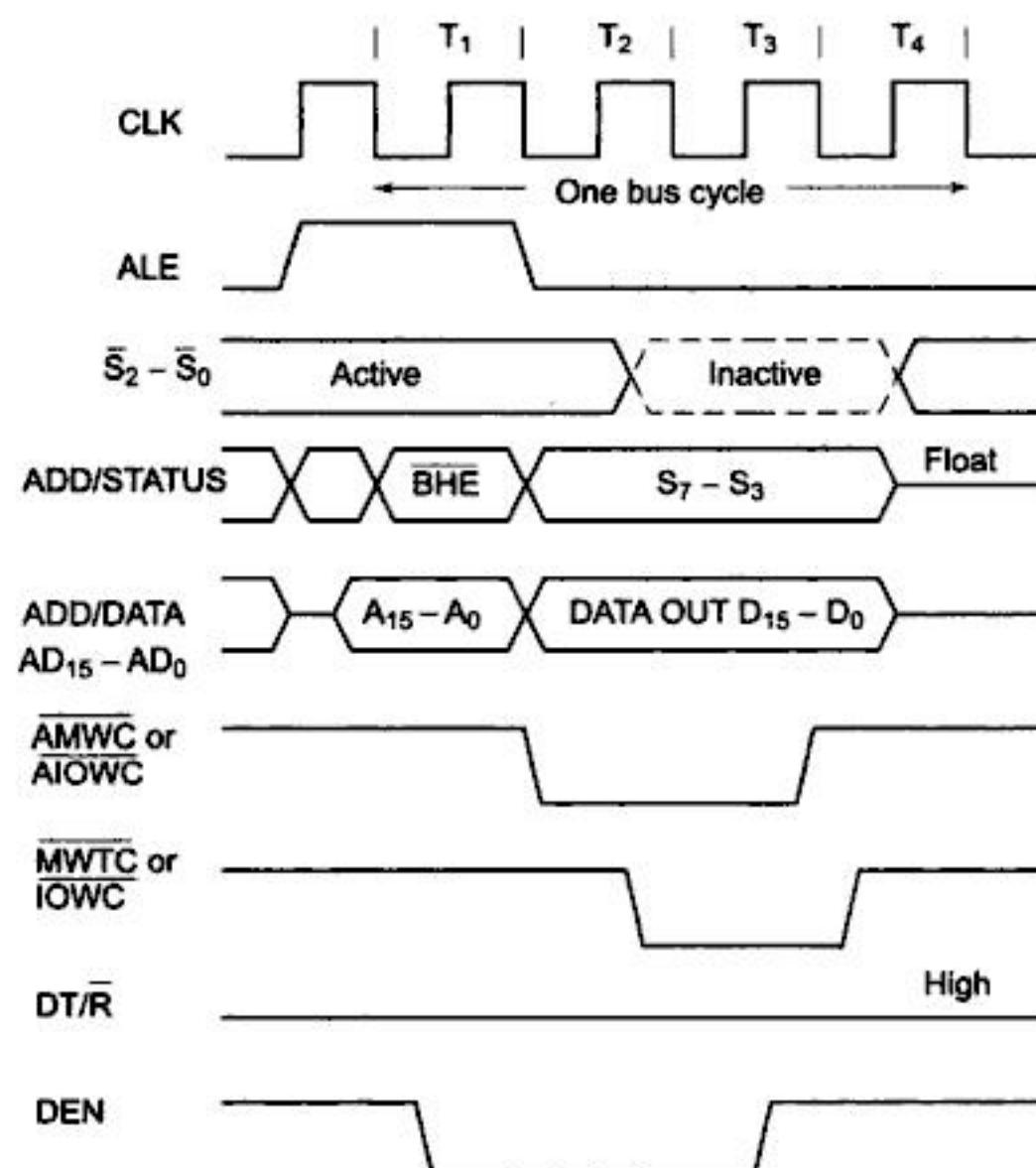
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

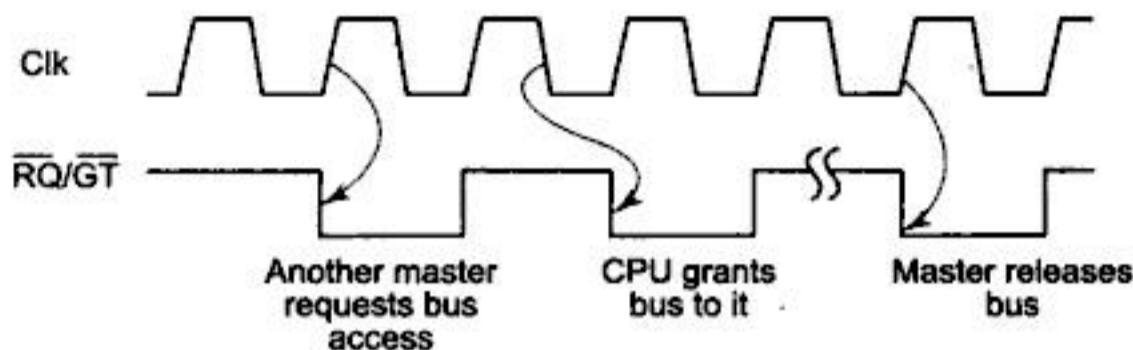


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Fig. 1.16 (a)** Memory Read Timing in Maximum Mode**Fig. 1.16(b)** Memory Write Timing in Maximum Mode

### 1.9.1 Timings for $\overline{RQ}/\overline{GT}$ Signals

The request/grant response sequence contains a series of three pulses as shown in the timing diagram Fig.1.16 (c). The request/grant pins are checked at each rising pulse of clock input. When a request is detected and if the conditions discussed in pin diagram section of this chapter for valid HOLD request are satisfied, the processor issues a grant pulse over the  $\overline{RQ}/\overline{GT}$  pin immediately during the  $T_4$  (current) or  $T_1$  (next) state. When the requesting master receives this pulse, it accepts the control of the bus. The requesting master uses the bus till it requires. When it is ready to relinquish the bus, it sends a release pulse to the processor (host) using the  $\overline{RQ}/\overline{GT}$  pin. This sequence is shown in Fig. 1.16 (c).



**Fig. 1.16 (c)  $\overline{RQ}/\overline{GT}$  Timings in Maximum Mode**

## 1.10 THE PROCESSOR 8088

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

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

### 1.10.1 Architecture and Signal Description of 8088

The register set of 8088 is exactly the same as that of 8086. The architecture of 8088 is also similar to 8086 except for two changes; a) 8088 has 4-byte instruction queue and b) 8088 has 8-bit data bus. The function of each block is the same as in 8086. Figure 1.17 shows the 8088 architecture.

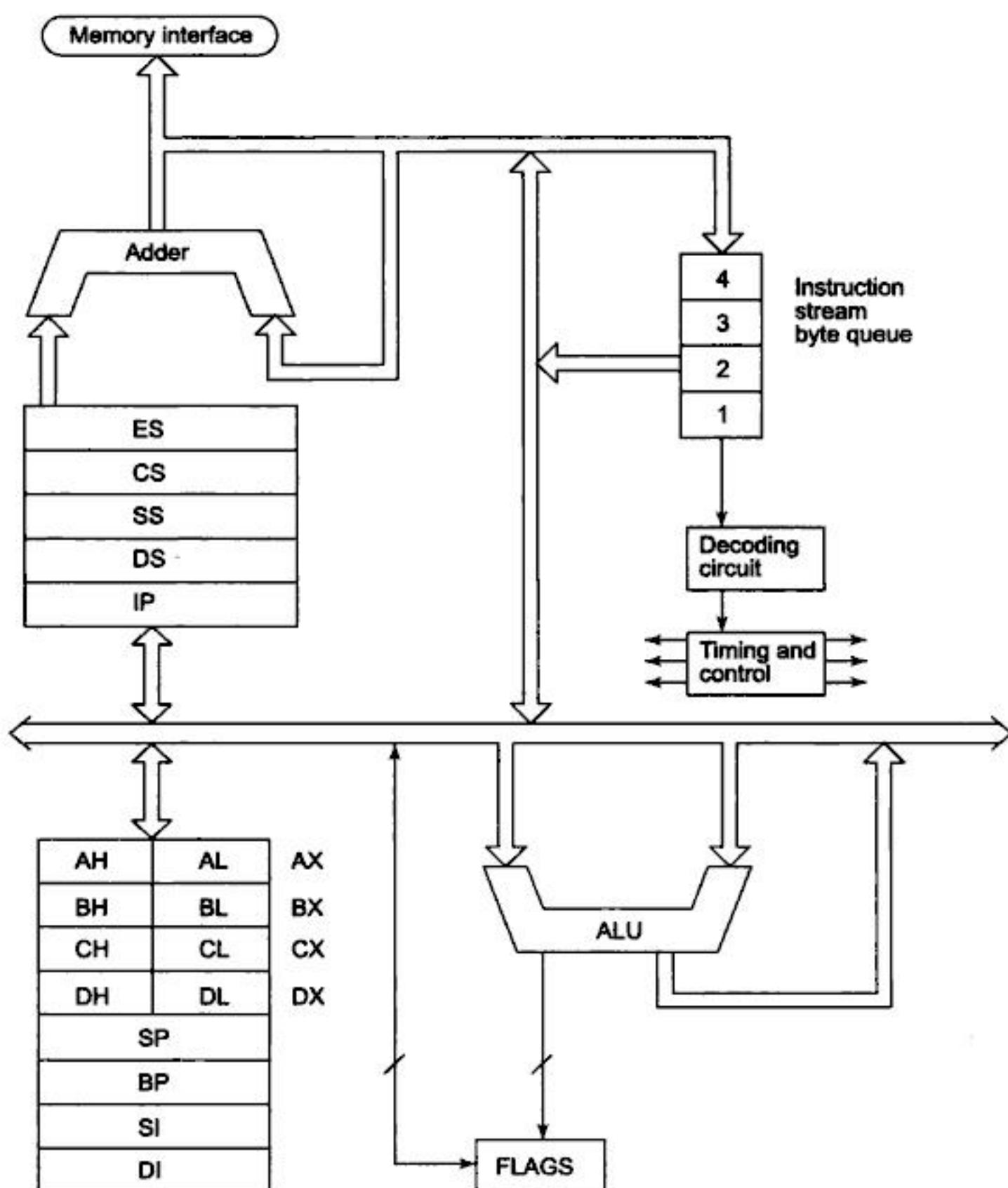


Fig. 1.17 Architecture of 8088

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

The pin diagram of 8088 is shown in Fig. 1.18. Most of the 8088 pins and their functions are exactly similar to the corresponding pins of 8086. Hence the pins that have different functions or timings are discussed in this section. Amongst them are the pins that have a common function in minimum and maximum mode.

	8088	Minimum mode	Maximum mode
GND	1	40	VCC
A <sub>14</sub>	2	39	A <sub>15</sub>
A <sub>13</sub>	3	38	A <sub>16/S<sub>3</sub></sub>
A <sub>12</sub>	4	37	A <sub>17/S<sub>4</sub></sub>
A <sub>11</sub>	5	36	A <sub>18/S<sub>5</sub></sub>
A <sub>10</sub>	6	35	A <sub>19/S<sub>6</sub></sub>
A <sub>9</sub>	7	34	SS <sub>0</sub> (High)
A <sub>8</sub>	8	33	MN/MX
AD <sub>7</sub>	9	32	RD
AD <sub>6</sub>	10	31	HOLD RQ/GT <sub>0</sub>
AD <sub>5</sub>	11	30	HLDA RQ/GT <sub>1</sub>
AD <sub>4</sub>	12	29	WR LOCK
AD <sub>3</sub>	13	28	IO/M S <sub>2</sub>
AD <sub>2</sub>	14	27	DT/R S <sub>1</sub>
AD <sub>1</sub>	15	26	DEN S <sub>0</sub>
AD <sub>0</sub>	16	25	ALE QS <sub>0</sub>
NMI	17	24	INTA QS <sub>1</sub>
INTR	18	23	TEST
CLK	19	22	READY
GND	20	21	RESET

Fig. 1.18 Pin Diagram of 8088

**AD<sub>7</sub>-AD<sub>0</sub> (Address/Data)** These lines constitute the address/data time multiplexed bus. During T<sub>1</sub> the bus is used for conducting addresses and during T<sub>2</sub>, T<sub>3</sub>, T<sub>w</sub> and T<sub>4</sub> states these lines are used for conducting data. These are tristated during 'hold acknowledge' and 'interrupt acknowledge' cycles.

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

**SS<sub>0</sub>** A new pin  $\overline{SS}_0$  is introduced in 8088 instead of  $\overline{BHE}$  pin in 8086. In minimum mode, the pin  $\overline{SS}_0$  is logically equivalent to the  $\bar{S}_0$  in the maximum mode. In maximum mode it is always high.

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

The signals  $\overline{SS}_0$ , DT/R, IO/M can be decoded to interpret the activities of the microprocessor as given in Table 1.6, in the minimum mode.

In the maximum mode, the pin  $\overline{SS}_0$  is permanently high. The functions and timings of other pins of 8088 are like that of 8086. Due to the difference in the bus structure, the timing diagrams are somewhat different.

**Table 1.6**

<b>IO/M</b>	<b>DT/R</b>	<b><math>\overline{SS}_0</math></b>	<b>Operation/Interpretation</b>
1	0	0	Interrupt Acknowledge
1	0	1	Read I/O port
1	1	0	Write I/O port
1	1	1	HALT
0	0	0	Code Access
0	0	1	Read memory
0	1	0	Write memory
0	1	1	Passive

### 1.10.2 Deriving 8088 Bus

As discussed earlier, 8088 is a microprocessor with an internal architecture, instruction set and memory addressing capability of 8086 but with the data bus of 8-bits like 8085. The 8-bit data bus makes 8088 compatible with the family of 8-bit peripherals designed around 8085.

For demultiplexing the bus, ALE signal is used to enable address latches. Usually three latch chips like 74373 are used for latching the twenty bit address while one bidirectional buffer chip (74245) is used for buffering the 8-bit data bus. The DEN and DT / R signals are used for buffering the data. The control bus may be derived exactly in the same way as that of 8086. The schematic diagram for deriving demultiplexed address/data bus is shown in Fig. 1.19, while the control bus can be derived as in Figs 1.20 (a) and 1.20 (b).

The minimum and maximum mode systems are also similar to the respective 8086 systems. The 8088 systems require only one data buffer due to the 8-bit data bus. The minimum and maximum mode systems of 8088 are shown in Figs 1.21 (a) and (b) respectively.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

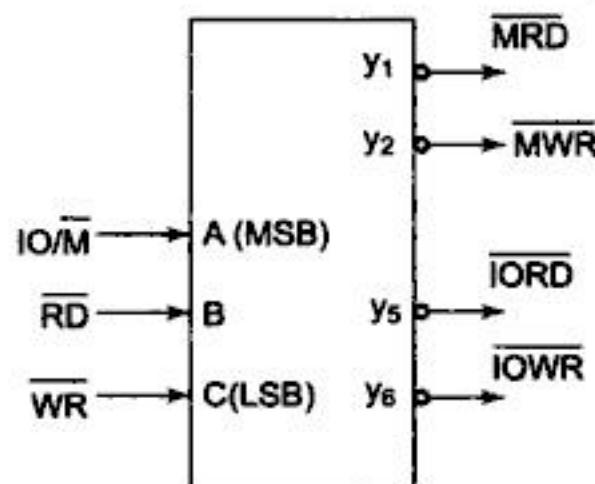


Fig 1.20 (b) Deriving 8088 Control Bus

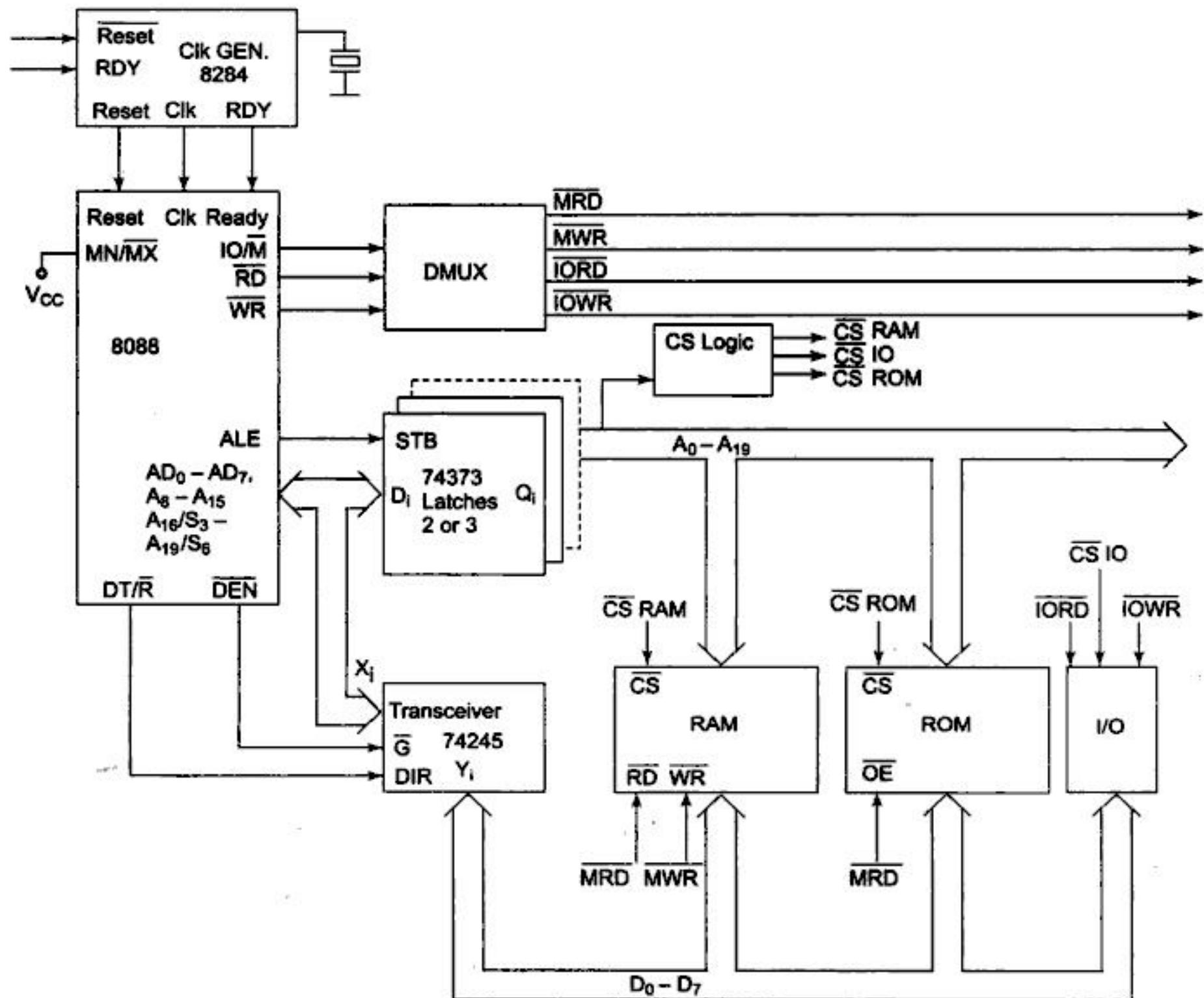


Fig 1.21 (a) Minimum Mode 8088 System

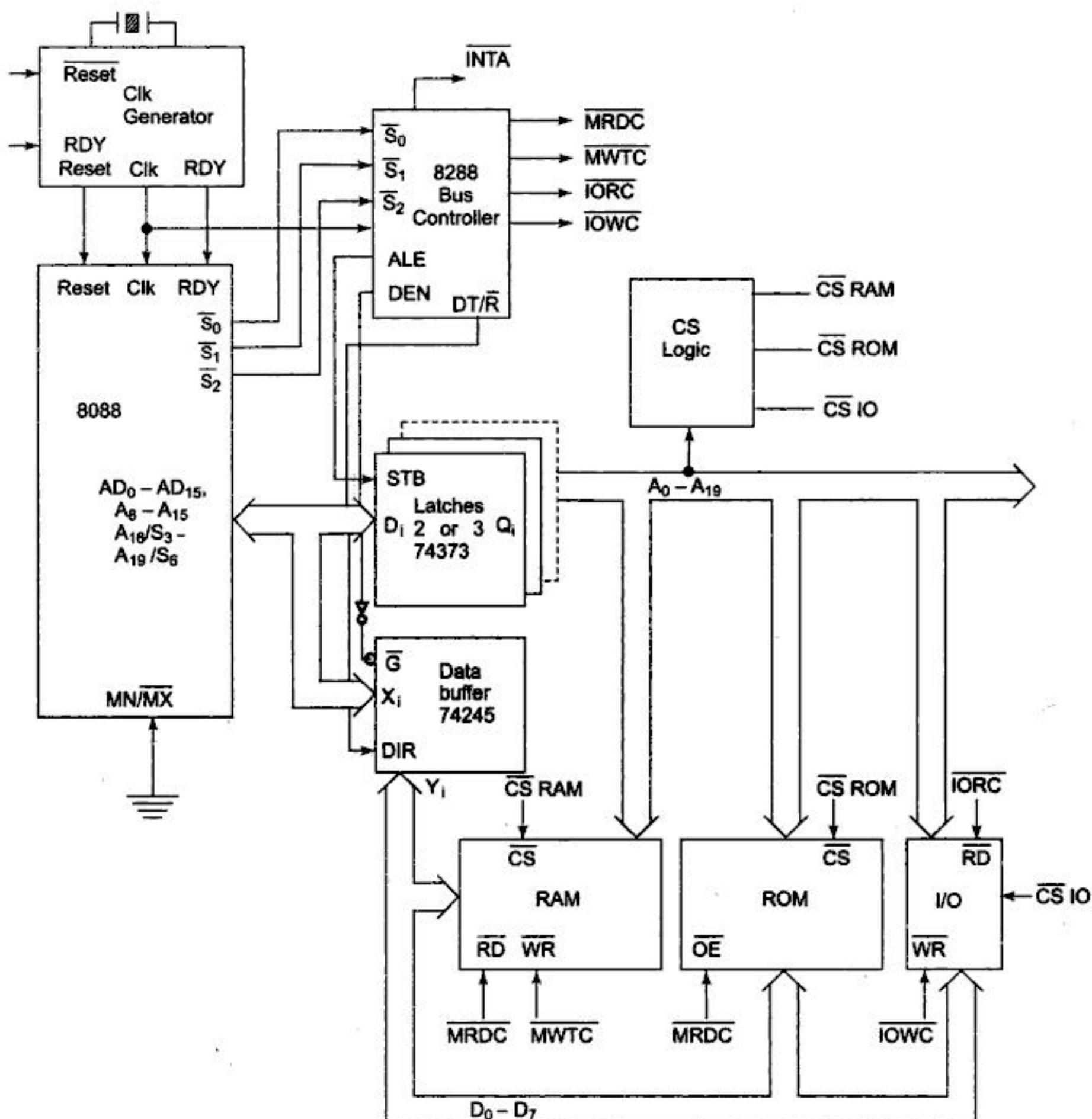


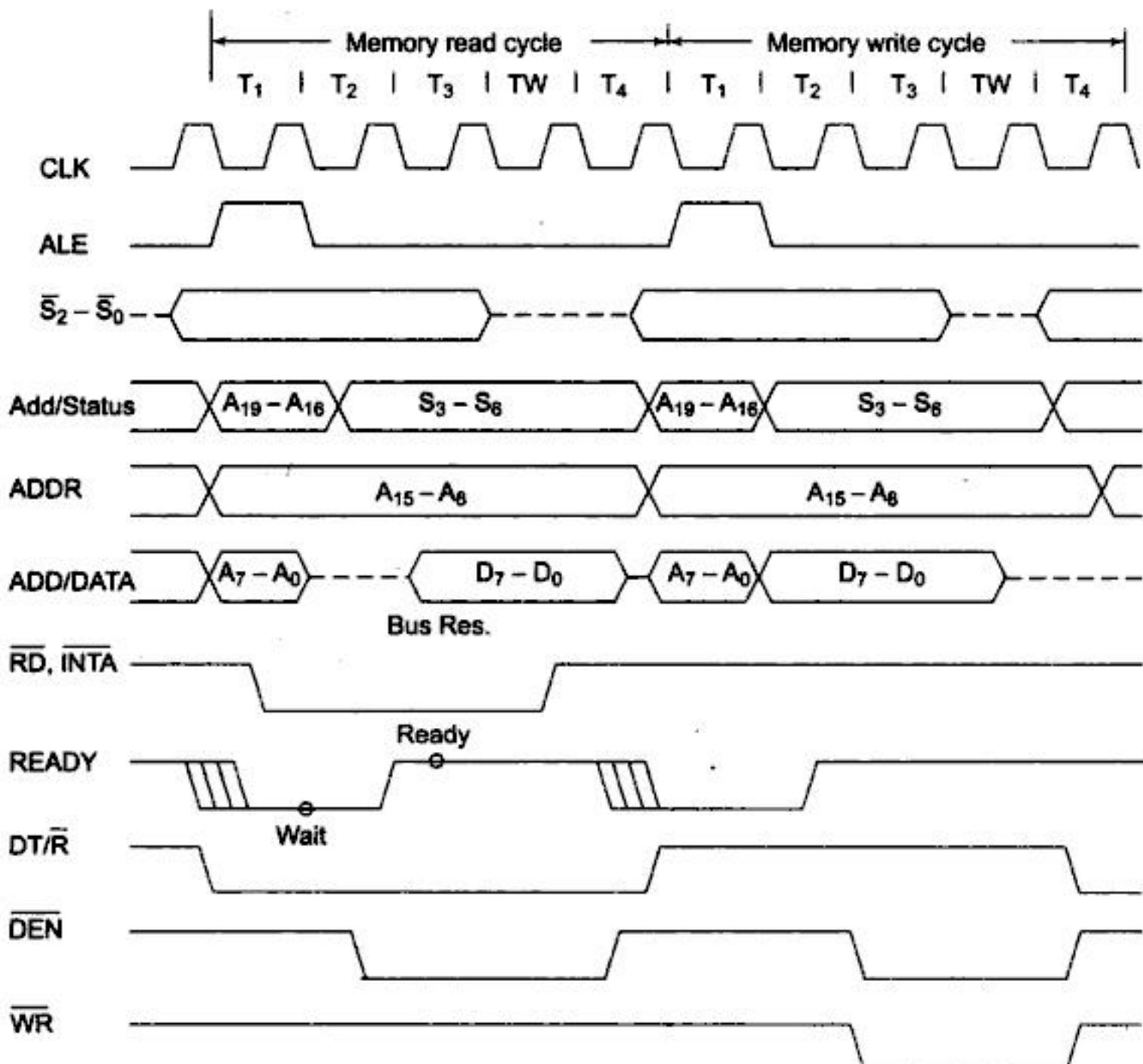
Fig 1.21 (b) Maximum Mode Minimum System of 8088

### 1.10.3 General 8088 System Timing Diagram

The 8088 address/data bus is divided into three parts (a) the lower 8 address/data bits, (b) the middle 8 address bits, and (c) the upper 4 address/status bits. The lower 8 lines are time multiplexed for address and data. The upper 4 lines are time multiplexed for address and status. Each of the bus cycles contains

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

In case of write cycle, the timing diagram is similar to the read cycle except for the validity of data. In write cycle, the data bits are available on the bus for  $T_2$ ,  $T_3$ ,  $T_w$ , and  $T_4$ . At the end of  $T_4$ , the bus is tristated. The other signals  $\overline{RD}$ ,  $\overline{WR}$ ,  $\overline{INTA}$ ,  $DT/\overline{R}$ ,  $\overline{DEN}$  and READY are similar to the 8086 timing diagram. Figure 1.22 shows the details of read and write bus cycles of 8088.



**Fig. 1.22** Read and Write Cycle Timing Diagram of 8088

#### 1.10.4 Comparison between 8086 and 8088

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

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

1. The predecoded code queue length is reduced to 4 bytes in 8088, whereas the 8086 queue contains 6 bytes. This was done to avoid the unnecessary prefetch operations and optimize the use of the bus by BIU while prefetching the instructions.
2. The 8088 bus interface unit will fetch a byte from memory to load the queue each time, if at least 1 byte is free. In case of 8086, at least 2 bytes should be free for the next fetch operation.
3. The overall execution time of the instructions in 8088 is affected by the 8-bit external data bus. All the 16-bit operations now require additional 4 clock cycles. The CPU speed is also limited by the speed of instruction fetches.

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

1.  $A_8 - A_{15}$  already latched, all time valid address bus.
2.  $\overline{BHE}$  has no meaning as the data bus is of 8-bits only.
3.  $\overline{SS}_0$  provides the  $S_0$  status information in minimum mode.
4.  $\overline{IO/M}$  has been inverted to be compatible with 8085 bus structure.

#### SUMMARY

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

#### Exercises

- 1.1 Draw and discuss the internal block diagram of 8086.
- 1.2 What do you mean by pipelined architecture? How is it implemented in 8086?

- 1.3 Explain the concept of segmented memory? What are its advantages?
- 1.4 Explain the physical address formation in 8086.
- 1.5 Draw the register organisation of 8086 and explain typical applications of each register.
- 1.6 Draw and discuss flag register of 8086 in brief.
- 1.7 Explain the function of the following signals of 8086.

(i) ALE	(ii) DT/R	(iii) DEN	(iv) LOCK
(v) TEST	(vi) MN/MX	(vii) BHE	(viii) M/IO
(ix) RQ/GT	(x) QS <sub>0</sub>	(xi) READY	(xii) NMI
(xiii) INTR	(xiv) HOLD	(xv) HLDA	

- 1.8 Explain the function of opcode prefetch queue in 8086.
- 1.9 How does 8086 differentiate between an opcode and instruction data?
- 1.10 Explain the physical memory organisation in an 8086 system.
- 1.11 What is the maximum memory addressing and I/O addressing capability of 8086?
- 1.12 Draw and discuss the read and write cycle timing diagrams of 8086 in minimum mode.
- 1.13 Draw and discuss the read and write cycle timing diagram of 8086 in maximum mode.
- 1.14 From which address the 8086 starts execution after reset?
- 1.15 How will you synchronise an external phenomenon like energising a relay with a program segment execution?
- 1.16 Draw and discuss a typical minimum mode 8086 system.
- 1.17 Draw and discuss a typical maximum mode 8086 system. What is the use of a bus controller in maximum mode?
- 1.18 Bring out the architectural and signal differences between 8086 and 8088.
- 1.19 What may be the reason for developing an externally 8-bit processor like 8088 after the 8086, when a 16-bit processor had already been introduced?
- 1.20 Explain the signal  $\bar{SS}_0$  of 8088.
- 1.21 Compare the bus interface of 8085 with 8088.
- 1.22 Draw and discuss a typical minimum mode 8088 system.
- 1.23 Draw and discuss a typical maximum mode 8088 system.
- 1.24 Draw and discuss a general 8088 system timing diagram.
- 1.25 What are the functions of the clock generator IC 8284, in the 8086/8088 systems?

# 8086/8088 Instruction Set and Assembler Directives

## INTRODUCTION

In Chapter 1, we have discussed the 8086/8088 architecture, pin diagrams and timing diagrams of read and write cycles. This chapter aims at introducing the readers with the general instruction formats, different addressing modes supported by 8086/8088 along with 8086/8088 instruction set. Further, a few important and frequently used assembler directives and operators have also been discussed. Thus this chapter creates a background for 'assembly language programming using 8086/8088'. A number of assemblers are available for programming with 8086/8088. Each of them has slightly different syntax, directives and operators. However, most of them work on similar principles. The directives and operators considered here are available with MASM (Microsoft MACRO ASSEMBLER).

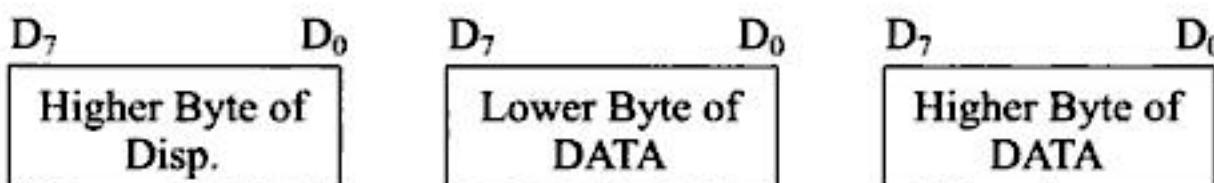
### 2.1 MACHINE LANGUAGE INSTRUCTION FORMATS

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

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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

**W-bit** This indicates whether the instruction is to operate over an 8-bit or 16-bit data/operands. If W bit is 0, the operand is of 8-bits and if W is 1, the operand is of 16-bits.

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

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

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

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

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

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

**Z-bit** This bit is used by REP instruction to control the loop. If Z bit is equal to 1, the instruction with REP prefix is executed until the zero flag matches the Z bit.

The REG code of the different registers (either as source or destination operands) in the opcode byte are assigned with binary codes. The segment registers are only 4 in number hence 2 binary bits will be sufficient to code them. The other registers are 8 in number, so at least 3-bits will be required for coding them. To allow the use of 16-bit registers as two 8-bit registers they are coded with W bit as shown in Table 2.1.

**Table 2.1 Assignment of Codes with Different Registers**

W bit	Register Address (code)	Registers	Segment 2 bit Register (code)	Segment Register
0	000	AL		
0	001	CL	00	ES
0	010	DL	01	CS
0	011	BL	10	SS
0	100	AH	11	DS
0	101	CH		
0	110	DH		
0	111	BH		
1	000	AX		
1	001	CX		
1	010	DX		
1	011	BX		
1	100	SP		
1	101	BP		
1	110	SI		
1	111	DI		

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

To find out the MOD and R/M fields of a particular instruction, one should first decide the addressing mode of the instruction. The addressing mode depends upon the operands and suggests how the effective address may be computed for locating the operand, if it lies in memory. The different addressing modes of the 8086 instructions are listed in Table 2.2. The R/M column and addressing mode row element specifies the R/M field, while the addressing mode column specifies the MOD field.

**Table 2.2 Addressing Modes and the Corresponding MOD, REG and R/M Fields**

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

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

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

DS is the default data segment register when a data is to be referred as an operand. CS is the default code segment register for storing program codes (executable codes). SS is the default segment register for the stack data accesses and operations. ES is the default segment register for the destination data storage. All the segments available (defined in a particular program) can be read or written as data segments by newly defining the data segment as required. There is no physical difference in the memory structure or no physical separation between the segment areas. They may or may not overlap with each other. Chapter 3 on 'Assembly Language Programming' explains the coding procedure of the instructions with suitable examples.

## 2.2 ADDRESSING MODES OF 8086

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

instruction execution, the instructions may be categorised as (i) Sequential control flow instructions and (ii) Control transfer instructions.

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

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

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

---

#### Example 2.1

MOV AX, 0005H

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

---

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

---

#### Example 2.2

MOV AX, [5000H]

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

---

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

---

#### Example 2.3

MOV BX, AX.

---

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

---

#### Example 2.4

MOV AX, [BX]

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

---

**5. Indexed** In this addressing mode, offset of the operand is stored in one of the index registers. DS is the default segment for index registers SI and DI. In case of string instructions DS and ES are default segments for SI and DI respectively. This mode is a special case of the above discussed register indirect addressing mode.

---

**Example 2.5**

MOV AX, [SI]

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

---

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

---

**Example 2.6**

MOV AX, 50H[BX]

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

---

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

---

**Example 2.7**

MOV AX, [BX] [SI]

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

---

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

---

**Example 2.8**

MOV AX, 50H [BX] [SI]

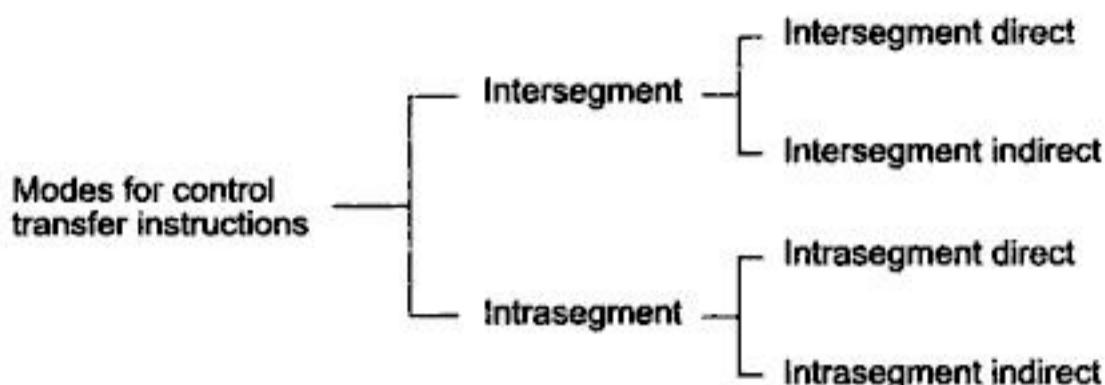
Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as  $10H \cdot DS + [BX] + [SI] + 50H$ .

---

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

*If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode. If the destination location lies in the same segment, the mode is called intrasegment mode.*

Figure 2.1 shows the modes for control transfer instructions.



**Fig. 2.1 Addressing Modes for Control Transfer Instructions**

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

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8 bits (i.e.  $-128 < d < +127$ ), we term it as *short jump* and if it is of 16 bits (i.e.  $-32768 < d < +32767$ ), it is termed as *long jump*.

#### Example 2.9

JMP SHORT LABEL; LABEL lies within -128 to +127 from the current IP content.  
Thus SHORT LABEL is 8-bit signed displacement.

A 16-bit target address of a label indicates that it lies within -32768 to +32767. But a problem arises when one requires a forward jump at a relative address greater than 32767 or backward jump at relative address -32768; in the same segment. Suppose current contents of IP are 5000H then a forward jump may be allowed at all the displacement DISP so that  $IP + DISP = FFFFH$  or  $DISP = FFFF - 5000 = AFFFH$ . Thus forward jumps may be allowed for all 16-bit displacement values from 0000H to AFFFH. If displacement exceeds AFFFH i.e. from B000H to FFFFH, then all such jumps will be treated as backward jumps. All such jumps are called NEAR PTR jumps and coded as below.

JMP NEAR PTR LABEL

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

#### Example 2.10

JMP [BX]; Jump to effective address stored in BX.  
JMP [ BX + 5000H ]

**11. Intersegment Direct** In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to

another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

#### Example 2.11

JMP 5000H : 2000H;  
Jump to effective address 2000H in segment 5000H.

**I2. Intersegment Indirect:** In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP(LSB), IP(MSB), CS(LSB) and CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

#### Example 2.12

JMP [2000H];  
Jump to an address in the other segment specified at effective address 2000H in DS, that points to the memory block as said above.

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

#### Example 2.13

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

Offset (displacement) = 5000H

[AX]-1000H, [BX]-2000H, [SI]-3000H, [DI]-4000H, [BP]-5000H,  
[SP]-6000H, [CS]-0000H, [DS]-1000H, [SS]-2000H, [IP]-7000H.

Shifting a number four times is equivalent to multiplying it by  $16_D$  or  $10_H$ .

(i) Direct addressing mode

MOV AX, [5000H]

$$\begin{array}{rcl} \text{DS:OFFSET} & \Leftrightarrow & 1000H:5000H \\ 10H^* DS & \Leftrightarrow & 10000 \\ \text{Offset} & \Leftrightarrow & +5000 \\ \hline & & 15000H - \text{Effective address} \end{array}$$

(ii) Register indirect

MOV AX, [BX]

$$\begin{array}{rcl} \text{DS:BX} & \Leftrightarrow & 1000H:2000H \\ 10H^* DS & \Leftrightarrow & 10000 \\ [\text{BX}] & \Leftrightarrow & +2000 \\ \hline & & 12000H - \text{Effective address} \end{array}$$

(iii) Register relative

MOV AX, 5000 [BX]

$$\begin{array}{rcl} \text{DS: } [5000 + \text{BX}] & & \\ 10H^* DS & \Leftrightarrow & 10000 \end{array}$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

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

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

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

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

### 2.3.1 Data Copy/Transfer Instructions

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

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

#### Example 2.16

Load DS with 5000H.

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

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

2. MOV AX, 5000H

MOV DS, AX

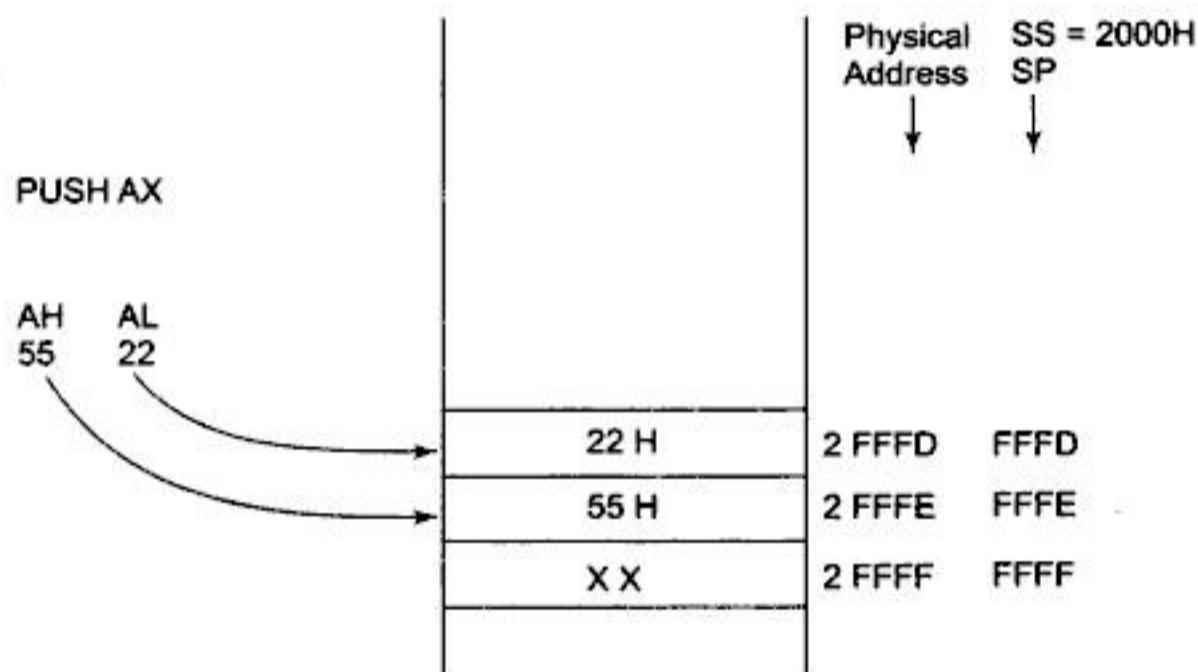
It may be noted here that both the source and destination operands cannot be memory locations (except for string instructions). Other MOV instruction examples are given below with the corresponding addressing modes.

3. MOV AX, 5000H;      Immediate
4. MOV AX, BX;      Register

- |                     |                                  |
|---------------------|----------------------------------|
| 5. MOV AX, [SI];    | Indirect                         |
| 6. MOV AX, [2000H]; | Direct                           |
| 7. MOV AX, 50H[BX]; | Based relative, 50H Displacement |

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

The actual operation takes place as given belows. SS : SP points to the stack top of 8086 system as shown in Fig. 2.2 and AH, AL contains data to be pushed.



**Fig. 2.2** Pushing data to stack memory

The sequence of operation as below:

1. Current stack top is already occupied so decrement SP by one then store AH into the address pointed to by SP.
2. Further decrement SP by one and store AL into the location pointed to by SP.

Thus SP is decremented by 2 and AH–AL contents are stored in stack memory as shown in Fig. 2.2. Contents of SP points to a new stack top.

The examples of these instructions are as follows:

#### Example 2.17

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

**POP: Pop from Stack** This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment

and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

16-bit contents of current stack top are popped into the specified operand as follows.

The sequence of operation is as below.

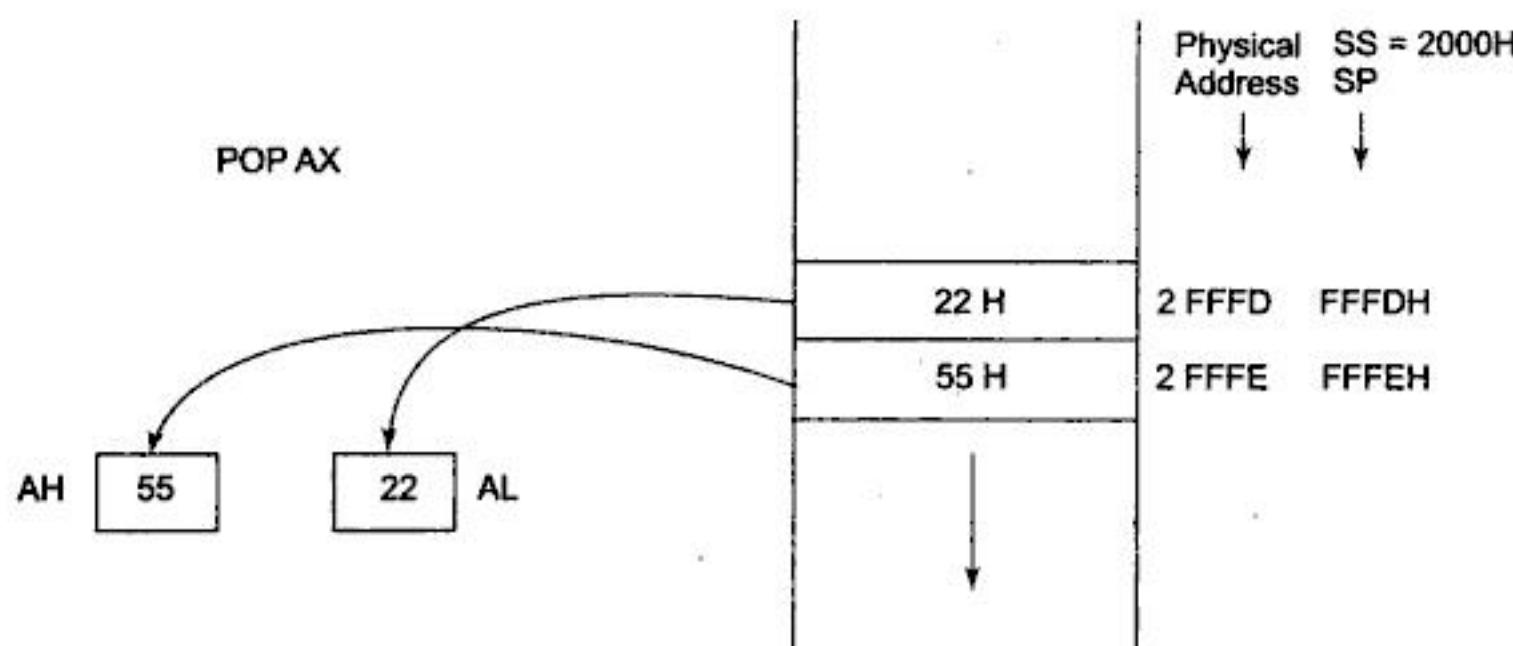
1. Contents of stack top memory location is stored in AL and SP is incremented by one
2. Further contents of memory location pointed to by SP are copied to AH and SP is again incremented by 1

Effectively SP is incremented by 2 and points to next stack top.

The examples of these instructions are shown as follows:

#### Example 2.18

1. POP AX
2. POP DS
3. POP [5000H]



**Fig. 2.3 Popping Register Contents from Stack Memory**

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

#### Example 2.19

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

**IN: Input the Port** This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly. AL and AX are the allowed destinations for 8

and 16-bit input operations. DX is the only register (implicit) which is allowed to carry the port address. If the port address is of 16 bits it must be in DX. The examples are given as shown:

#### Example 2.20

1. IN AL,03H ; This instruction reads data from an 8-bit port whose address is 03H and stores it in AL.
2. IN AX, DX ; This instruction reads data from a 16-bit port whose address is in DX (implicit) and stores it in AX.
3. MOV DX,0800H ; The 16-bit address is taken in DX.  
IN AX, DX ; Read the content of the port in AX.

**OUT: Output to the Port** This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D<sub>8</sub>-D<sub>15</sub> while that to an even addressed port is transferred on D<sub>0</sub>-D<sub>7</sub>. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively. If the port address is of 16 bits it must be in DX. The examples are given as shown:

#### Example 2.21

1. OUT 03H,AL ; This sends data available in AL to a port whose address is 03H.
2. OUT DX, AX ; This sends data available in AX to a port whose address is specified implicitly in DX.
3. MOV DX, 0300H ; The 16-bit port address is taken in DX.  
OUT DX, AX ; Write the content of AX to a port of which address is in DX.

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

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

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

<i>Mnemonics &amp; Description</i>		<i>Instruction Code</i>		
Register/Memory	1111111 w	mod 000 r/m		
Register	01000 reg			
<b>AAA = ASCII Adjust for Addition</b>	00110111			
<b>DAA = Decimal Adjust for Addition</b>	00100111			
<b>SUB = Subtract</b>				
Reg/Memory and Register to Either	001010 dw	mod reg r/m		
Immediate from Register/Memory	100000 sw	mod 101 r/m	data	data if s w = 01
Immediate from Accumulator	0010110 w	data	data if w = 1	
<b>SBB = Subtract with Borrow</b>				
Reg/Memory and Register to Either	000110 dw	mod reg r/m		
Immediate from Register/Memory	100000 sw	mod 011 r/m	data	data if s w = 01
Immediate from accumulator	0001110 w	data	data if w = 1	
<b>DEC = Decrement:</b>				
Register/Memory	1111111 w	mod 001 r/m		
Register	01001 reg			
<b>NEG = Change sign</b>	1111011 w	mod 011 r/m		
<b>CMP = Compare:</b>				
Register/Memory and Register	001110 dw	mod reg r/m		
Immediate with Register/Memory	100000 sw	mod 111 r/m	data	data if s w = 01
Immediate with Accumulator	0011110 w	data	data if w = 1	
<b>AAS = ASCII Adjust for Subtract</b>	00111111			
<b>DAS = Decimal Adjust for Subtract</b>	00101111			
<b>MUL = Multiply (Unsigned)</b>	1111011 w	mod 100 r/m		
<b>IMUL = Integer Multiply (Signed)</b>	1111011 w	mod 101 r/m		
<b>AAM = ASCII Adjust Multiply</b>	11010100	00001010		
<b>DIV = Divide (Unsigned)</b>	1111011 w	mod 110 r/m		
<b>IDIV = Integer Divide (Signed)</b>	1111011 w	mod 111 r/m		
<b>AAD = ASCII Adjust for Divide</b>	11010101	00001010		
<b>CBW = Convert Byte to Word</b>	10011000			
<b>CWD = Convert Word to Double Word</b>	10011001			
<b>LOGICAL</b>	76543210	76543210	76543210	76543210
<b>NOT = Invert</b>	1111011 w	mod 010 r/m		
<b>SHL/SAL = Shift Logical/Arithmetic Left</b>	110100 v w	mod 100 r/m		
<b>SHR = Shift Logical Right</b>	110100 v w	mod 101 r/m		
<b>SAR = Shift Arithmetic Right</b>	110100 v w	mod 111 r/m		
<b>ROL = Rotate Left</b>	110100 v w	mod 000 r/m		
<b>ROR = Rotate Right</b>	110100 v w	mod 001 r/m		
<b>RCL = Rotate Through Carry Flag Left</b>	110100 v w	mod 010 r/m		
<b>RCR = Rotate Through Carry Right</b>	110100 v w	mod 011 r/m		
<b>AND = And:</b>				
Reg/Memory and Register to Either	001000 dw	mod reg r/m		
Immediate to Register/Memory	1000000 w	mod 100 r/m	data	data if w = 1
Immediate to Accumulator	0010010 w	data	data if w = 1	
<b>TEST = And Function to Flags, No Result:</b>				
Register/Memory and Register	1000010 w	mod reg r/m		

(Contd.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Mnemonics & Description		Instruction Code
<b>JP/JPE</b> = Jump on Parity/Parity Even	01111010	disp
<b>JO</b> = Jump on Overflow	01110000	disp
<b>JS</b> = Jump on Sign	01111000	disp
<b>JNE/JNZ</b> = Jump on Not Equal/Not Zero	01110101	disp
<b>JNL/JGE</b> = Jump on Not Less/Greater or Equal	01111101	disp
<b>JNLE/JG</b> = Jump on Not Less or Equal/Greater	01111111	disp
<b>JNB/JAE</b> = Jump on Not Below/Above or Equal	01110011	disp
<b>JNBE/JA</b> = Jump on Not Below or Equal/Above	01110111	disp
<b>JNP/JPO</b> = Jump on Not Par/Par Odd	01111011	disp
<b>JNO</b> = Jump on Not Overflow	01110001	disp
<b>JNS</b> = Jump on Not Sign	01111001	disp
<b>LOOP</b> = Loop CX Times	11100010	disp
<b>LOOPZ/LOOPE</b> = Loop While Zero/Equal	11100001	disp
<b>LOOPNZ/LOOPNE</b> = Loop While Not Zero/Equal	11100000	disp
<b>JCXZ</b> = Jump on CX Zero	11100011	disp
<b>INT</b> = Interrupt		
<b>Type Specified</b>	11001101	type
<b>Type 3</b>	11001100	
<b>INTO</b> = Interrupt on Overflow	11001110	
<b>IRET</b> = Interrupt Return	11001111	
	76543210	76543210
<b>PROCESSOR CONTROL</b>		
<b>CLC</b> = Clear Carry	11111000	
<b>CMC</b> = Complement Carry	11110101	
<b>STC</b> = Set Carry	11111001	
<b>CLD</b> = Clear Direction	11111100	
<b>STD</b> = Set Direction	11111101	
<b>CLI</b> = Clear Interrupt	11111010	
<b>STI</b> = Set Interrupt	11111011	
<b>HLT</b> = Halt	11110100	
<b>WAIT</b> = Wait	10011011	
<b>ESC</b> = Escape (to External Device)	11011xxx	mod xxx r/m
<b>LOCK</b> = Bus Lock Prefix	11110000	

\*The v, w, d, s and z bits and the mod, reg, r/m fields are discussed in the addressing modes' section.

Fig. 2.4 8086/8088 Instruction Set Summary

**Example 2.22**


---

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

---

**LEA: Load Effective Address** The load effective address instruction loads the effective address formed by destination operand into the specified source register. This instruction is more useful for assembly language rather than for machine language. The examples are given below.

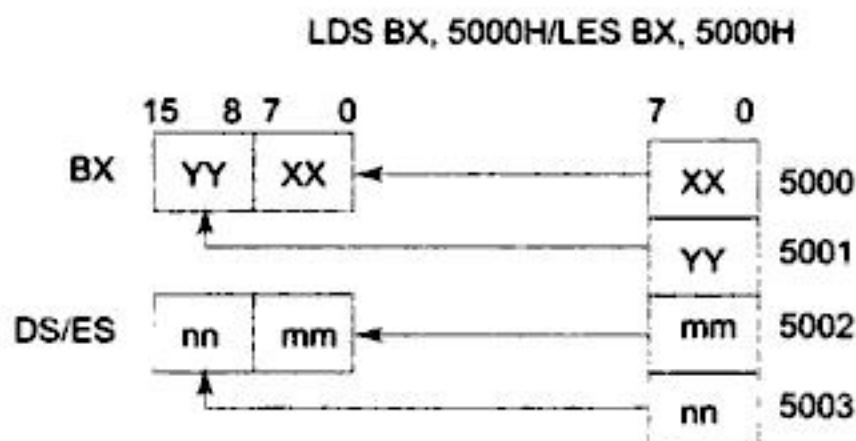
**Example 2.23**


---

LEA BX,ADR	; Effective address of Label ADR i.e. offset of ADR will be transferred to Reg ; BX.
LEA SI, ADR[Bx]	; offset of Label ADR will be added to content of Bx to form effective ; address and it will be loaded in SI

---

**LDS/LES: Load Pointer to DS/ES** This instruction loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction. The example in Fig. 2.5 explains the operation.



**Fig. 2.5 LDS/LES Instruction Execution**

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

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

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

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

Figure 2.4 shows the data sheet for the hand coding of all the 8086 instructions. The MOD and R/M fields are to be decided as already described in this chapter. This type of instructions do not affect any flags.

### 2.3.2 Arithmetic Instructions

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

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

---

#### Example 2.24

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

---

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

---

#### Example 2.25

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

---

**INC: Increment** This instruction increases the contents of the specified register or memory location by 1. All the condition code flags are affected except the carry flag CF. This instruction adds 1 to the

contents of the operand. Immediate data cannot be operand of this instruction. The examples of this instruction are as follows:

---

**Example 2.26**

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

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

---

**Example 2.27**

1. DEC AX Register
  2. DEC [5000H] Direct
- 

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

---

**Example 2.28**

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

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

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

---

**Example 2.29**

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

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

#### Example 2.30

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

**AAA: ASCII Adjust After Addition** The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4 high order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4 bits of AL are cleared and AH is incremented by one. If the value in the lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by 1, the AF and CF flags are set to 1, and the higher 4 bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Fig. 2.6. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

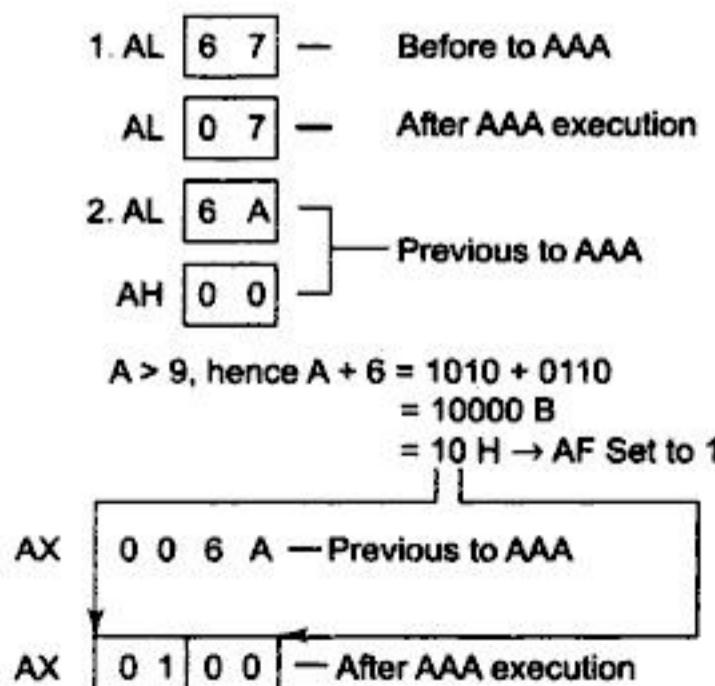


Fig. 2.6 ASCII Adjust after Addition Instruction

**AAS: ASCII Adjust AL after Subtraction** AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4 bits

of AL register are greater than 9 or if the AF flag is 1, the AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9. The procedure is similar to the AAA instruction except for the subtraction of 06 from AL. AH is modified as difference of the previous contents (usually zero) of AH and the borrow for adjustment.

**AAM : ASCII Adjust after Multiplication** This instruction, after execution, converts the product available in AL into unpacked BCD format. The AAM—ASCII Adjust After Multiplication—instruction follows a multiplication instruction that multiplies two unpacked BCD operands, i.e. higher nibbles of the multiplication operands should be 0. The multiplication of such operands is carried out using MUL instruction. Obviously the result of multiplication is available in AX. The following AAM instruction replaces content of AH by tens of the decimal multiplication and AL by singles of the decimal multiplication.

---

#### Example 2.31

```
MOV AL, 04 ; AL ← 04
MOV BL, 09 ; BL ← 09
MUL BL ; AH-AL ← 24H (9 × 4)
AAM ; AH ← 03
; AL ← 06
```

---

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

---

#### Example 2.32

AX	05	08
----	----	----

AAD result in AL [00] 3A 58D = 3A H in AL

---

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

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

**Example 2.33**

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

(ii) AL = 73                  CL = 29  
   ADD AL, CL                ; AL  $\leftarrow$  AL + CL  
                               ; AL  $\leftarrow$  73 + 29  
                               ; AL  $\leftarrow$  9C  
   DAA                        ; AL  $\leftarrow$  02 and CF = 1  
                               AL = 7 3  
                               +  
                               CL = 2 9  
                               9 C  
                               + 6  
                                      
                               A 2  
                               + 6 0  
                                      
                               CF = 1 0 2 in AL

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

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

**Example 2.34**

(i) AL = 75                  BH = 46  
   SUB AL,BH                ; AL  $\leftarrow$  2 F = (AL) - (BH)  
                               ; AF = 1  
   DAS                        ; AL  $\leftarrow$  2 9 (as F > 9, F - 6 = 9)  

(ii) AL = 38                  CH = 6 1  
   SUB AL,CH                ; AL  $\leftarrow$  D 7 CF = 1 (borrow)  
   DAS                        ; AL  $\leftarrow$  7 7 (as D > 9, D - 6 = 7)  
                               ; CF = 1 (borrow)

DAA and DAS instructions are also called packed BCD arithmetic instructions.

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

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

---

#### Example 2.35

1. MUL BH ; (AX)  $\leftarrow$  (AL)  $\times$  (BH)
  2. MUL CX ; (DX) (AX)  $\leftarrow$  (AX)  $\times$  (CX)
  3. MUL WORD PTR [SI] ; (DX) (AX)  $\leftarrow$  (AX)  $\times$  ([SI])
- 

**IMUL: Signed Multiplication** This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF, and ZF flags are undefined after IMUL. If AH and DX contain parts of 16 and 32-bit result respectively, CF and OF both will be set. The AL and AX are the implicit operands in case of 8 bits and 16 bits multiplications respectively. The unused higher bits of the result are filled by sign bit and CF, AF are cleared. The example instructions are given as follows:

---

#### Example 2.36

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

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

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

**DIV: Unsigned Division** This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) and an interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The

quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

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

### 2.3.3 Logical Instructions

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

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

#### Example 2.37

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

If the content of AX is 3F0FH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register.

0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1	= 3F0F H [AX]
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	AND
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008 H
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008 H [AX]

The result 0008H will be in AX.

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

#### Example 2.38

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

The contents of AX are say 3F0FH, then the first example instruction will be carried out as given below.

0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1	= 3F0F H
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	OR
0 0 0 0	0 0 0 0	1 0 0 1	1 0 0 0	= 0098 H
0 0 1 1	1 1 1 1	1 0 0 1	1 1 1 1	= 3F9F H

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

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

#### Example 2.39

NOT AX

NOT [5000H]

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

AX =	0 0 1 0	0 0 0 0	0 0 0 0	1 1 1 1
invert	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓
	1 1 0 1	1 1 1 1	1 1 1 1	0 0 0 0

Result

in AX = D F F 0

The result DFF0H will be stored in the destination register AX.

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

#### Example 2.40

1. XOR AX, 0098H

2. XOR AX, BX

3. XOR AX, [5000H]

If the content of AX is 3F0FH, then the first example instruction will be executed as explained.

The result 3F97H will be stored in AX.

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

**TEST: Logical Compare Instruction** The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to 1, if the corresponding bits of both operands are 1, else the result bit is reset to 0. The result of this ANDing operation is not available for further use, but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

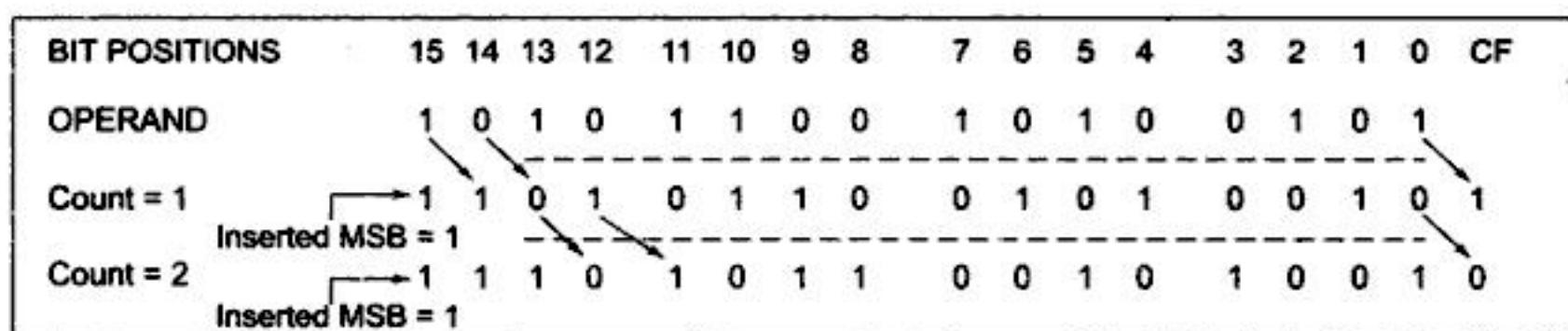


Fig. 2.9 Execution of SAR Instruction

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

**ROR: Rotate Right without Carry** This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it cannot be an immediate operand. Figure 2.10 explains the operation. The destination operand may be a register (except a segment register) or a memory location.

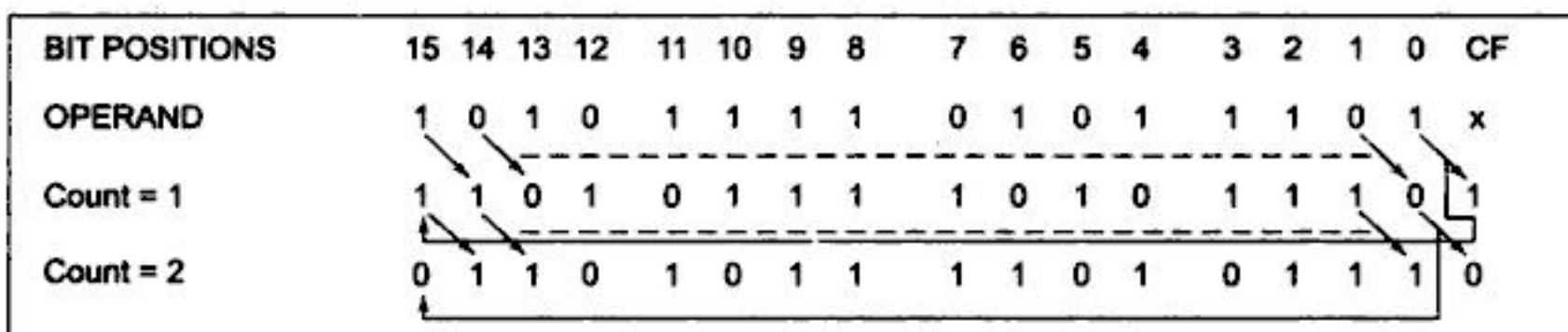


Fig. 2.10 Execution of ROR Instruction

**ROL: Rotate Left without Carry** This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF, and ZF flags are left unchanged in this rotate operation. The operand may be a register or a memory location. Figure 2.11 explains the operation.

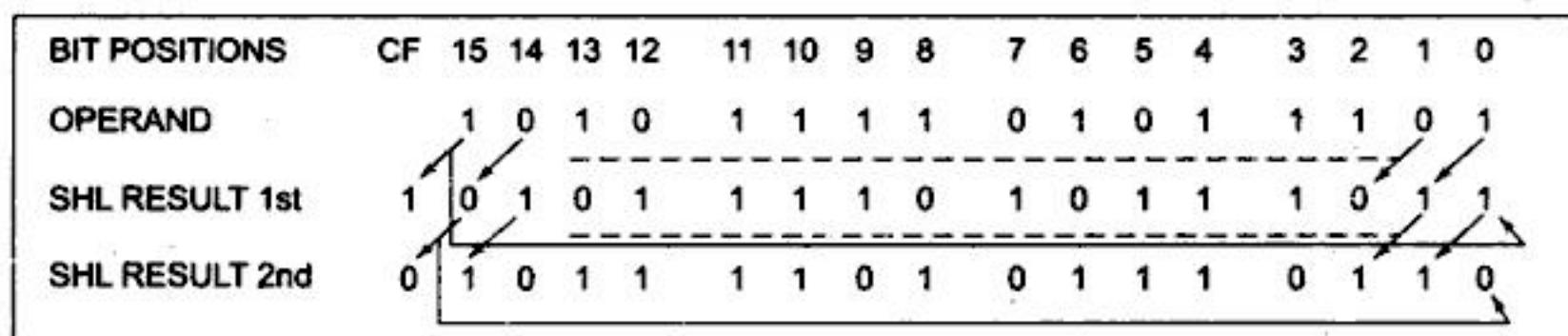


Fig. 2.11 Execution of ROL Instruction

**RCR: Rotate Right through Carry** This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF). For each operation, the carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 2.12 explains the operation.

BIT POSITIONS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CF (arbitrary)
OPERAND	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1	0
Count = 1					0	1	1	1	1	0	1	0	1	1	1	0	1

Fig. 2.12 Execution of RCR Instruction

**RCL: Rotate Left through Carry** This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB , and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 2.13 explains the operation.

BIT POSITIONS	CF (arbitrary)	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPERAND	0	1	0	0	1	1	1	0	1	1	0	1	1	0	1	0	1
Count = 1		1	0	0	1	1	1	0	1	1	0	1	1	0	1	0	1

Fig. 2.13 Execution of RCL Instruction

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

### 2.3.4 String Manipulation Instructions

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as *byte strings* or *word strings*. For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. For referring to a string, two parameters are required, (a) starting or end address of the string and (b) length of the string. The length of a string is usually stored as count in the CX register. In case of 8085, similar structures can be set up by the pointer and counter arrangements which may be modified at each iteration, till the required condition for proceeding further is satisfied. On the other hand, the 8086 supports a set of more powerful instructions for string manipulations. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the Direction Flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two. The counter in both the cases, is decremented by one.

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

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

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

---

#### Example 2.42

```

MOV AX,5000H      ; Source segment address is 5000h
MOV DS,AX         ; Load it to DS
MOV AX,6000H      ; Destination segment address is 6000h
MOV ES,AX         ; Load it to ES
MOV CX,0FFH       ; Move length of the string to counter register CX
MOV SI,1000H       ; Source index address 1000H is moved to SI
MOV DI,2000H       ; Destination index address 2000H is moved to DI
CLD              ; Clear DF, i.e. set autoincrement mode
REP MOVSB        ; Move OFFH string bytes from source address to destination

```

---

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

The following string of instructions explain the instruction. The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the direction flag and the counter is decremented. This byte by byte or word by word comparison

continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

#### Example 2.43

```

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

```

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

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

#### Example 2.44

```

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

```

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

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

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

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each iteration automatically. If DF = 1, then the execution follows autodecrement mode. In this mode, SI and DI are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in autodecrementing mode, the strings are referred to by their ending addresses. If DF = 0, then the execution follows autoincrement mode. In this mode, SI and DI are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses. Chapter 3 on assembly language programming explains the use of some of these instructions in assembly language programs.

### 2.3.5 Control Transfer or Branching Instructions

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. Depending upon the addressing modes specified in Chapter 1, the CS may or may not be modified. This type of instructions are classified in two types:

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

**Conditional Control Transfer (Branch) Instructions** In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. The results of the previous operations are replicated by condition code flags. In other words, using this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

### 2.3.6 Unconditional Branch Instructions

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Figure 2.14 shows the arrangement of CS and IP addresses of the ISR in the interrupt vector table.

Memory Contents	15	8 7	0	15	8 7	0
	CS High			IP High		
	CS Low			IP Low		
CS High	0000	:	0083			
CS Low	0000	:	0082			
IP High	0000	:	0081			
IP Low	0000	:	0080			

**Fig. 2.14** *Contents of IVT*

**INTO: Interrupt on Overflow** This command is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0010 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

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

JUMP	DISP 8-bit	Intrasegment, relative, short jump			
JUMP	DISP.16-bit (LB)	DISP.16-bit (UB)	Intrasegment, relative, short jump		
JUMP	IP (LB)	IP (UB)	CS (LB)	CS (UB)	Intrasegment, direct, far jump

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

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

#### Example 2.46

```
MOV CX, 0005 ; Number of times in CX
MOV BX, OFF7H ; Data to BX
```

```

Label : MOV AX, CODE1
        OR BX, AX
        AND DX, AX
Loop   Label

```

The execution proceeds in sequence, after the loop is executed, CX number of times. If CX is already 00H, the execution continues sequentially. No flags are affected by this instruction.

### 2.3.7 Conditional Branch Instructions

When these instructions are executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied. If not the execution continues sequentially. The conditions, here, means the status of condition code flags. These type of instructions do not affect any flag. The address has to be specified in the instruction relatively in terms of displacement which must lie within -80H to 7FH (or -128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it lies within the above specified range. The different 8086/8088 conditional branch instructions and their operations are listed in Table 2.3.

**Table 2.3 Conditional Branch Instructions**

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

Another type of hint which helps the assembler to assign a particular constant with a label or initialise particular memory locations or labels with constants is an *operator*. In fact, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler or Turbo Assembler. The directives and operators are discussed here but their meanings and uses will be more clear in Chapter 3 on assembly language programming techniques.

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

#### Example 2.47

```
RANKS    DB 01H, 02H, 03H, 04H
```

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

```
MESSAGE DB 'GOOD MORNING'
```

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

```
VALUE    DB 50H
```

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

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

#### Example 2.48

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

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialisation, the lower bytes are stored at the lower

memory addresses, while the upper bytes are stored at the higher addresses. Another option of the DW directive is explained with the DUP operator.

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

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

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

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

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

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

**ENDP: END of Procedure** In assembly language programming, the subroutines are called procedures. They may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP. The statements, appearing in the same module but after the ENDP directive, are neglected from that procedure. The structure given below explains the use of ENDP.

PROCEDURE STAR
STAR ENDP

**ENDS: END of Segment** This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments. Whatever are the contents of the segments,

they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment. The structure shown below explains the fact more clearly.

DATA	SEGMENT
	:
DATA	ENDS
ASSUME	CS : CODE, DS : DATA
CODE	SEGMENT
	:
CODE	ENDS
END	

The above structure represents a simple program containing two segments named DATA and CODE. The data related to the program must lie between the DATA SEGMENT and DATA ENDS statements. Similarly, all the executable instructions must lie between CODE SEGMENT and CODE ENDS statements.

**EVEN: Align on Even Memory Address** The assembler, while starting the assembling procedure of any program, initialises a location counter and goes on updating it, as the assembly proceeds. It goes on assigning the available addresses, i.e. the contents of the location counter, sequentially to the program variables, constants and modules as per their requirements, in the sequence in which they appear in the program. The EVEN directive updates the location counter to the next even address, if the current location counter contents are not even, and assigns the following routine or variable or constant to that address. The structure given below explains the directive.

EVEN	ROOT
PROCEDURE	
	:
ROOT	ENDP

The above structure shows a procedure ROOT that is to be aligned at an even address. The assembler will start assembling the main program calling ROOT. When the assembler comes across the directive EVEN, it checks the contents of the location counter. If it is odd, it is updated to the next even value and then the ROOT procedure is assigned to that address, i.e. the updated contents of the location counter. If the content of the location counter is already even, then the ROOT procedure will be assigned with the same address.

**EQU: Equate** The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitutes the numerical value for that label and finds out the equivalent code. Using the EQU directive, even an instruction mnemonic can be assigned with a label, which can then be used in the program in place of that mnemonic. Suppose, a numerical constant which appears in a program ten times. If that constant is to be changed at a later time, one will have to make the correction 10 times. This may lead to human errors, because it is possible that a human programmer may miss one of those corrections. This will



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

```
CONTINUE LABEL FAR
```

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

DATA	SEGMENT
DATAS	DB 50H DUP (?)
DATA-LAST	LABEL BYTE FAR
DATA	ENDS

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

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

```
MOV CX, LENGTH ARRAY
```

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

**LOCAL** The labels, variables, constants or procedures declared LOCAL in a module are to be used only by that particular module. After some time, some other module may declare a particular data type LOCAL, which was previously declared LOCAL by an other module or modules. Thus the same label may serve different purposes for different modules of a program. With a single declaration statement, a number of variables can be declared local, as shown.

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

**NAME: Logical Name of a Module** The NAME directive is used to assign a name to an assembly language program module. The module, may now be referred to by its declared name. The names, if selected to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

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

#### Example 2.50

```
CODE SEGMENT
MOV SI, OFFSET LIST
CODE ENDS
```

---

```
DATA SEGMENT
LIST DB 10H
DATA ENDS
```

---

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

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

---

#### Example 2.51

RESULT	PROC	NEAR
ROUTINE	PROC	FAR

---

**PTR: Pointer** The POINTER operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16-bit quantity. In other words, the PTR operator is used to specify the data type—byte or word. The examples of the PTR operator are as follows:

---

#### Example 2.52

MOV AL, BYTE PTR [SI] -	Moves content of memory location addressed by SI (8-bit) to AL
INC BYTE PTR [BX]-	Increments byte contents of memory location addressed by BX
MOV BX, WORD PTR [2000H]-	Moves 16-bit content of memory location 2000H to BX, i.e. [2000H] to BL [2001H] to BH
INC WORD PTR [3000H] -	Increments word contents of memory location 3000H considering contents of 3000H (lower byte) and 3001H (higher byte) as a 16-bit number

---

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far, as explained in the examples given below.

```
JMP NEAR PTR [BX]-NEAR Jump
JMP FAR PTR [BX]-FAR Jump
```

**PUBLIC** As already discussed, the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive. On the other hand, the data types declared EXTRN in a module of the program, may be declared PUBLIC in at least any one of the other modules of the same program. (Refer to the explanation on EXTRN directive to get the clear idea of PUBLIC.)

**SEG: Segment of a Label** The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of "SEG" label. The example given below explains the use of SEG operator.

#### Example 2.53

```
MOV AX, SEG ARRAY ; This statement moves the segment address of ARRAY in
MOV DS, AX          ; which it is appearing, to register AX and then to DS.
```

**SEGMENT: Logical Segment** The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program. In some cases, the segment may be assigned a type like PUBLIC (i.e. can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules). The program structure given below explains the use of the SEGMENT directive.

```
EXE.CODE SEGMENT GLOBAL      ; Start of segment named EXE.CODE,
                             ; that can be accessed by any other module.
EXE.CODE ENDS                ; END of EXE.CODE logical segment.
```

**SHORT** The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within -128 to +127 bytes from the address of the byte next to the jump opcode). This method of specifying the jump address saves the memory. Otherwise, the assembler may reserve two bytes for the displacement. The syntax of the statement is as given below.

```
JMP SHORT LABEL
```

**TYPE** The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE' label by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction MOV AX, TYPE STRING moves the value 0002H in AX.

**GLOBAL** The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

```
ROUTINE PROC GLOBAL
```

**' + & -' Operators** These operators represent arithmetic addition and subtraction respectively and are typically used to add or subtract displacements (8 or 16 bit) to base or index registers or stack or base pointers as given in the example:

---

**Example 2.54**

```
MOV AL, [ SI +2 ]
MOV DX, [ BX - 5 ]
MOV BX, [ OFFSET LABEL + 10 H ]
MOV AX, [ BX + 9I ]
```

---

**FAR PTR** This directive indicates the assembler that the label following FAR PTR is not available within the same segment and the address of the label is of 32-bits i.e. 2 bytes offset followed by 2 bytes segment address.

---

**Example 2.55**

```
JMP FAR PTR LABEL
CALL FAR PTR ROUTINE
```

---

Both the above instructions indicate to the assembler that the target address is going to require four bytes; Lower byte of offset, higher byte of offset, lower byte of segment and higher byte of segment; indicating intersegment addressing mode.

**NEAR PTR** This directive indicates that the label following NEAR PTR is in the same segment and needs only 16 bit i.e. 2 byte offset to address it.

---

**Example 2.56**

```
JMP NEAR PTR LABEL
CALL NEAR PTR ROUTINE
```

---

If a label is not preceded by NEAR PTR or FAR PTR, then it is by default considered a NEAR PTR label and two bytes are reserved by the assembler for its address during the process of assembling.

---

**SUMMARY**

This chapter is aimed at introducing the readers with the instruction set of 8086/88 and the most commonly used assembler directives and operators. To start with, the available instruction formats in 8086/88 instruction set are explained in details. Further, the addressing modes available in 8086/88 are discussed in significant details with necessary examples. The 8086/88 instructions can be broadly categorized in six types depending upon their functions, namely data transfer instructions, arithmetic instructions and logical instructions, shift and rotate instructions, string manipulation instructions, control transfer instructions and processor control instructions. All these instruction types have been discussed before proceeding with the assembler directives and operators. The

*coding information details of all these instructions may be obtained from the Intel data sheets. The necessary assembler directives have been discussed later with their possible syntax, functions and examples. Most of these directives are available in Microsoft MASM. The detailed discussion on every assembler directive and operator is out of the scope of this book. The readers may refer to MASM Programmer's Guide and Technical reference for further details of the directives and operators, discussed in this chapter. Chapter 3 elaborates on how to use the instructions, assembler directives and operators to write assembly language programs.*

## Exercises

- 2.1 State and explain the different instruction formats of 8086/8088.
- 2.2 What do you mean by addressing modes? What are the different addressing modes supported by 8086? Explain each of them with suitable examples.
- 2.3 Explain the physical address formation in different addressing modes.
- 2.4 Explain the addressing modes for control transfer instructions.
- 2.5 What are the different instruction types of 8086?
- 2.6 'A single instruction may use more than one addressing mode or some instructions may not require any addressing mode'. Explain.
- 2.7 Bring out the developments in 8086 instruction set over 8085 instruction set, in details.
- 2.8 Explain the execution of all the instructions of 8086 with suitable examples.
- 2.9 What are the assembler directives and pseudo-ops?
- 2.10 Explain all the assembler directives, pseudo-ops and operators presented in this chapter with suitable examples.
- 2.11 How does the CPU identify between 8-bit and 16-bit operations?
- 2.12 How is the addressing mode of an instruction communicated to the CPU?
- 2.13 What is the difference between the jump and loop instructions?
- 2.14 Which instruction of 8086 can be used for look up table manipulations?
- 2.15 What is the difference between the respective shift and rotate instructions?
- 2.16 How will you enter the single step mode of 8086?
- 2.17 What is LOCK prefix? What is its use?
- 2.18 What is REP prefix? What is its use?

# The Art of Assembly Language Programming with 8086/8088

## INTRODUCTION

In the previous chapter, the 8086/8088 instruction set and assembler directives were discussed in significant detail. This chapter aims at making the reader more familiar with the instructions and assembler directives and their use in implementing the different structures required for the implementation of algorithms. In this chapter, the different structures are implemented by using the instruction set of 8086. A number of example programs are discussed to explain the use of these structures. While in the second chapter, a qualitative study of all the addressing modes has been presented, in this chapter, the ideas about the addressing modes and their typical uses will be presented more clearly through example programs. After studying this chapter, one will be in a position to use the instructions and directives properly to translate an algorithm into a program. While emphasizing on different programming techniques, we have stressed more on managing the processor resources and capabilities because while solving a particular problem, the programmer may find a number of solutions (instruction sequences). A skilled programmer selects an optimum solution out of them for that specific application. For example, the instruction INC AL and ADD AL,01H may serve the same purpose but the first one requires less memory and execution time than the second one. Hence, the INC instruction will be preferred over ADD. Also the improper use of general purpose and special purpose registers may lead to the requirement of more instructions for a particular algorithm resulting in more execution time and memory requirement. While implementing



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The above instruction sequence is quite straightforward. As the immediate data cannot be loaded into a segment register, the data is transferred to one of the general purpose registers, say AX, and then the register content is moved to the segment register DS. Thus the data segment register DS contains 2000H. The instruction MOV AX,[500H] signifies that the contents of the particular location, whose offset is specified in the brackets with the segment pointed to by DS as segment register, is to be moved to AX. The MOV [0700H],AX instruction moves the contents of the register AX to an offset 0700H in DS (DS = 2000H). Note that the code segment register CS gets automatically loaded by the code segment address of the program whenever it is executed. Actually it is the monitor program that accepts the CS:IP address of the program and passes it to the corresponding registers at the time of execution. Hence no instructions like DS or SS are required for loading the CS register.

### Example 3.2

Write a program to move the contents of the memory location 0500H to register BX and to CX. Add immediate byte 05H to the data residing in memory location, whose address is computed using DS=2000H and offset=0600H. Store the result of the addition in 0700H. Assume that the data is located in the segment specified by the data segment register DS which contain 2000H.

**Solution** The flow chart for the program is shown in Fig. 3.2.

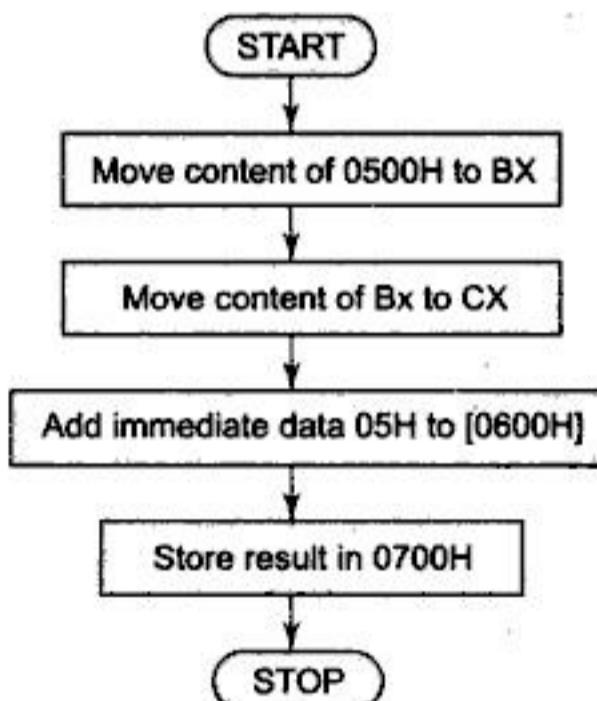
```

MOV  AX, 2000H
MOV  DS, AX      ; Initialize data segment
                  ; register
MOV  BX, [0500H] ; Get contents of 0500H in BX
MOV  CX, BX      ; Copy the same contents in CX
ADD  [0600H], 05H ; Add byte 05H to contents of
                  ; 0600H
MOV  DX, [0600H] ; Store the result in DX
MOV  [0700H], DX  ; Store the result in 0700H
HLT              ; Stop

```

After initialising the data segment register, the content of location 0500H are moved to the BX register using MOV instruction. The same data is moved also to the CX register. For this data transfer, there may be two options as shown.

- (a) MOV CX, BX ; As the contents of BX will be  
; same as 0500H after execution  
; of MOV BX, [0500H].
- (b) MOV CX, [0500H] ; Move directly from 0500H to register CX



**Fig. 3.2 Flow Chart for Example 3.2**

The *opcode* in the first option is only of 2 bytes, while the second option will have 4 bytes of *opcode*. Thus the second option will require more memory and execution time. Due to these reasons, the first option is preferable.

The immediate data byte 05H is added to the content of 0600H using the ADD instruction. The result will be in the destination operand 0600H. This is next stored at the location 0700H. In case of the 8086/8088 instruction set, there is no instruction for the direct transfer of data from the memory source operand to the memory destination operand except, the string instructions. Hence the result of addition

which is present at 0600H, should be moved to any one of the general purpose registers, except BX and CX, otherwise the contents of CX and BX will be changed. We have selected DX (we could have selected AX also, because once DS is initialised to 2000H the contents of AX are no longer useful) for this purpose. Thus the transfer of result from 0600H to 0700H is accomplished in two stages using successive MOV instructions, i.e. at first, the content of 0600H is moved to DX and then the content of DX is moved to 0700H. The program ends with the HLT instruction.

### Example 3.3

Add the contents of the memory location 2000H:0500H to contents of 3000H:0600H and store the result in 5000H:0700H.

**Solution** Unlike the previous example programs, this program refers to the memory locations in different segments, hence, while referring to each location, the data segment will have to be newly initialised with the required value. Figure 3.3 shows the flow chart.

The instruction sequence for the above flow chart is given along with the comments.

```

MOV CX, 2000H      ; Initialize DS at 2000H
MOV DS, CX
MOV AX, [500H]      ; Get first operand in AX
MOV CX, 3000H      ; Initialize DS at 3000H
MOV DS, CX
MOV BX, [0600H]      ; Get second operand in BX.
ADD AX, BX          ; Perform addition
MOV CX, 5000H      ; Initialize DS at 5000H
MOV DS, CX
MOV [0700H], AX     ; Store the result of addition in
                    ; 0700H and stop
HLT
    
```

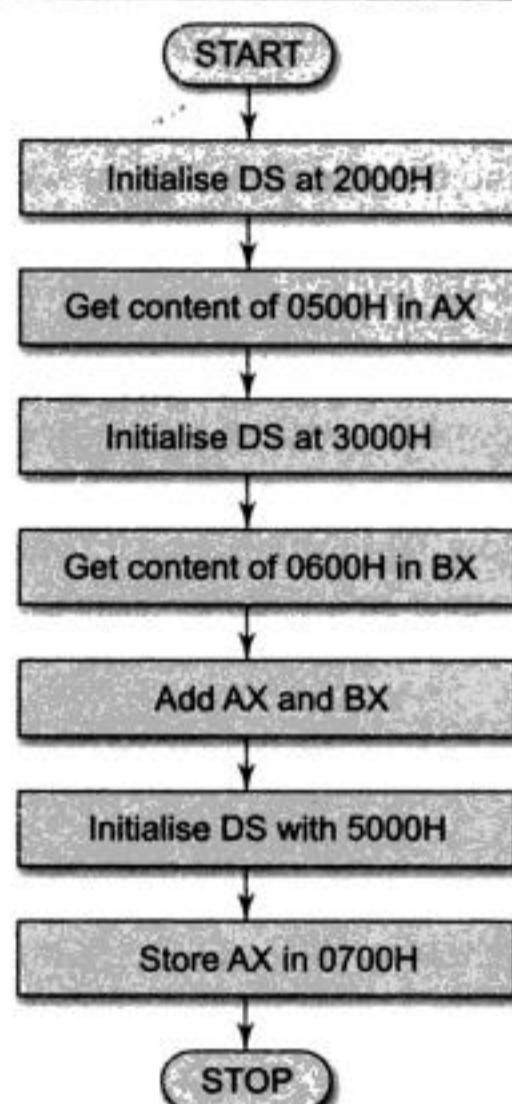


Fig. 3.3 Flow Chart for Example 3.2

Actually, the program simply performs the addition of two operands which are located in different memory segments. The program has become lengthy only due to data segment register initialization instructions.

### Example 3.4

Move a byte string, 16-bytes long, from the offset 0200H to 0300H in the segment 7000H.

**Solution** According to the program statement, a string that is 16-bytes long is available at the offset address 0200H in the segment 7000H. The required program should move this complete string at offset 0300H, in the same segment. Let us emphasize this program in the light of comparison between 8085 and 8086 programming techniques.

An 8085 program to perform this task, is given neglecting the segment addresses.

---

MVI C, 010H	; Count for the length of string
LXI H 0200H	; Initialization of HL pair for source string
LXI D 0300H	; Initialization of DE pair for destination
BACK : MOV A, M	; Take a byte from source in A
STAX D	; Store contents of A to address pointed to by DE pair
INX H	; Increment source pointer
INX D	; Increment destination pointer
DCRC	; Decrement counter
JNZ BACK	; Continue if counter is not zero
HLT	; Stop if counter is zero

---

The programmers, fluent with 8085 assembly language programming but starting with 8086, may translate the above 8085 assembly language program listings to 8086 assembly language programs using the analogous or comparable instructions. Of course, this method of programming is not efficient, however, it may help those who are familiar to 8085 programming and wish to start writing programs in 8086 assembly language. The reason for the inefficiency of this method is that the special features and capabilities of 8086 have not been taken into account while preparing the 8086 assembly language program.

Now, let us think about how the above program may be transferred to 8086 assembly language using analogous instructions. Note that the segment initialization is to be added. Let us consider that the code and data segment address is 7000H. Consider that the code starts at offset 0000H.

MOV AX, 7000H	
MOV DS, AX	; Data segment initialization
MOV SI, 0200H	; Pointer to source string
MOV DI, 0300H	; Pointer to destination string
MOV CX, 0010H	; Count for length of string
BACK : MOV AL, [SI]	; Take a source byte in AL
MOV [DI], AL	; Move it to destination
INC SI	; Increment source pointer
INC DI	; Increment destination pointer
DEC CX	; Decrement count by 1
JNZ BACK	; Continue if count is not 0
HLT	; Stop if the count is 0

The above list has been prepared using the program written in 8085 ALP. Indexed addressing mode is used for string byte accesses and transfer in this case. The functions of all the 8086 instructions and the 8086 addressing modes have already been explained in Chapter 2. In this program, all the instructions used are more or less analogous to the 8085 program, and the special software capabilities of 8086 like string instructions and loop instructions have not been considered. The 8086 programs based on 8085 codes are inefficient due to the reason that the full capability of the rich 8086 instruction set and the enhanced architecture of 8086 cannot be fully exploited.

The above program uses the decrement and jump-if-not-zero instructions for checking whether the transfer is complete or not. The 8086 instruction set provides LOOP instructions for this purpose. Using these instructions, the program is modified as shown:

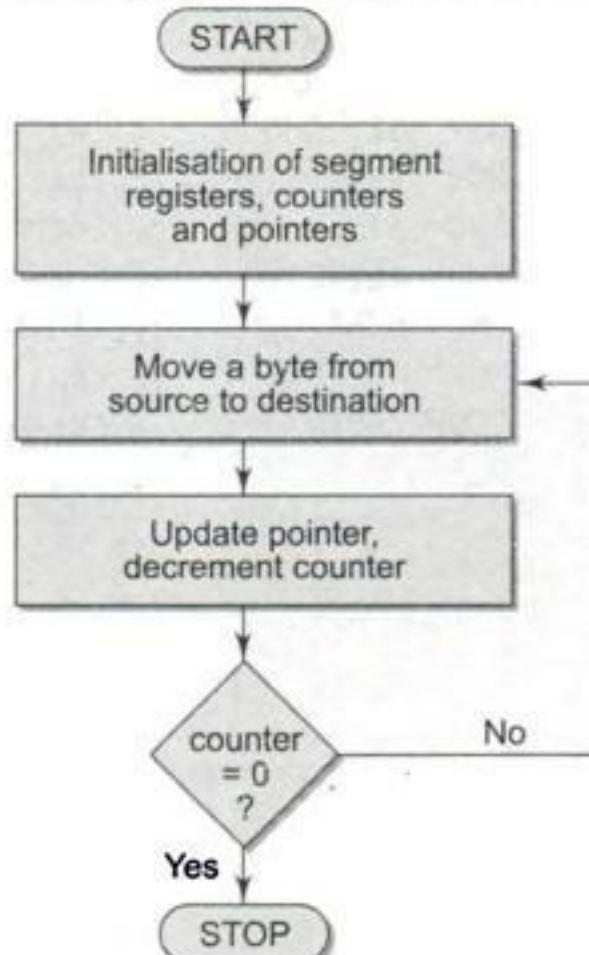
```

MOV AX,7000H
MOV DS,AX      ; Data segment initialization
MOV SI,0200H   ; Source pointer initialization
MOV DI,0300H   ; Destination pointer initialization
MOV CX,0010H   ; Counter initialization
BACK : MOV AL,[SI]    ; Take a byte of string from source
        MOV [DI],AL   ; and then move it to destination
        INC SI        ; Update source pointer
        INC DI        ; Update destination pointer continue
        LOOP BACK     ; till CX=0,[DEC CX and JNZ BACK]
        HLT          ; Stop if CX=0

```

Thus the two instructions bracketed in the comment field are replaced by a single loop instruction which results in the saving of memory and execution time. The loop instruction needs the additional instructions for updating the pointers (for example, INC SI, INC DI). It does not need counter decrement and check-if-zero instruction.

One more feature of the 8086 instruction set is the string instruction, i.e **MOVSB** and **MOVSW**. Using these instructions one can move a string byte/word from source to destination. The length of the string is specified by the CX register. The SI and DI point to the source and destination locations. The DS and ES registers should be initialised to source and destination segment addresses respectively. Before the use of string instructions, the program should initialise all these registers properly. Using the string byte instruction the same program may be written as shown:



**Fig. 3.4 Flow Chart for Example 3.4**

```

MOV AX, 7000H
MOV DS, AX      ; Source segment initialisation
MOV ES,AX       ; Destination segment initialisation
MOV CX,0010H   ; Counter initialisation
MOV SI,0200H   ; Source pointer initialisation
MOV DI,0300H   ; Destination pointer initialisation
CLD           ; Clear DF
REP MOVSB      ; Move the complete string
HLT          ; Stop

```

The **MOVSB** instruction needs neither counter decrement and jump back nor pointer update instructions. All these functions are done automatically. An experienced programmer will thus directly use the string instructions instead of using other options. The flow chart of the final program is presented in Fig. 3.4.

**Example 3.5**

Find out the largest number from an unordered array of sixteen 8-bit numbers stored sequentially in the memory locations starting at offset 0500H in the segment 2000H.

**Solution** The logic for this procedure can be described as follows. The first number of the array is taken in a register, say AL. The second number of the array is then compared with the first one. If the first one is greater than the second one, it is left unchanged. However, if the second one is greater than the first, the second number replaces the first one in the AL register. The procedure is repeated for every number in the array and thus it requires 15 iterations. At the end of 15th iteration the largest number will reside in the register AL. This may be represented in terms of the flow chart as shown in Fig. 3.5. The listing is given below:

```

MOV CX, 0F H      ; Initialize counter for number of iterations
MOV AX, 2000H     ; Initialize data segment
MOV DS, AX         ;
MOV SI, 0500H     ; Initialize source pointer
MOV AL, [SI]       ; Take first number in AL
BACK : INC SI      ; Increment source pointer
            CMP AL,[SI]   ; Compare next number with the previous
            JNC NEXT      ; If the next number is larger
            MOV AL,[SI]   ; replace the previous one with the next
NEXT : LOOP BACK    ; Repeat the procedure 15 times
            HLT

```

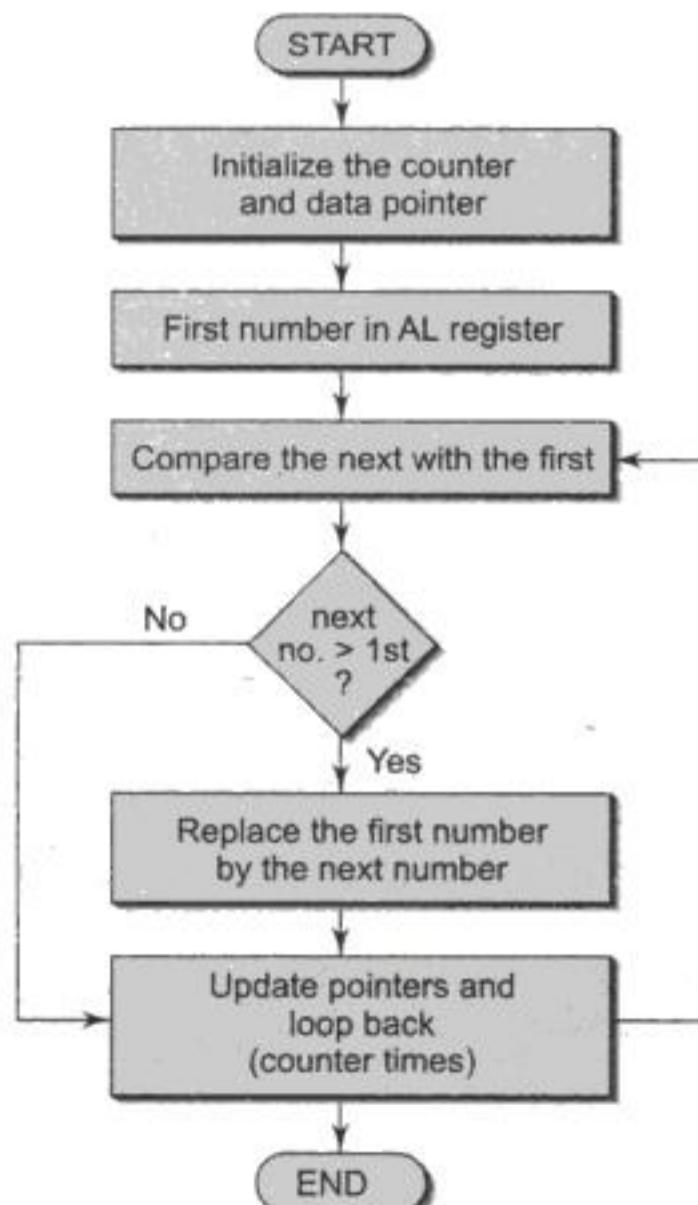


Fig. 3.5 Flow Chart for Example 3.5



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

D <sub>7</sub> D <sub>6</sub> D <sub>5</sub> D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub> D <sub>1</sub> D <sub>0</sub>	D <sub>7</sub>	D <sub>0</sub>	D <sub>7</sub>	D <sub>0</sub>
1 0 1 1	W	REG	DATA LOW BYTE		DATA HIGH BYTE	

Following the procedure as in Example 3.6, the code comes out to be (BB 00 50) as shown.

CODE	W	REG	Data LB	Data HB
1 0 1 1	1	0 1 1	0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 0
B	B		0 0	5 0

### Example 3.8

MOV [SI], DL

This instruction belongs to the register to memory format. Hence from the data sheet Fig. 2.4 and using the already explained procedure, the machine code can be found as shown.

OPCODE	D	W	MOD	REG	R/M
1 0 0 0 1 0	0	1	0 0	0 1 0	1 0 0
8	9		1		4

The machine code is 89 14.

### Example 3.9

MOV BP[SI], 0005H

OPCODE	W	MOD	OPCODE	R/M
1 1 0 0 0 1 1	1	0 0	0 0 0	0 1 0
C 7		0		2

LOWER BYTE	HIGHER BYTE
0 0 0 0 1 0 1	0 0 0 0 0 0 0
0 5	0 0

The machine code of this instruction is C7 02 05 00.

### Example 3.10

MOV BP [SI+ 500H], 7293H

OPCODE	W	MOD	OPCODE	R/M
1 1 0 0 0 1 1	1	1 0	0 0 0	0 1 0
C 7		8		2

LOWER BYTE DISP.	HIGHER BYTE DISP.
0 0 0 0 0 0 0	0 0 0 0 0 1 0 1
0 0	0 5

Displacement 500H

LOWER BYTE DATA	HIGHER BYTE DATA
1 0 0 1 0 0 1 1	0 1 1 1 0 0 1 0
9 3	7 2

Data 7293 H

The complete machine code comes out to be C7 82 00 05 93 72.

**Example 3.11**

ADD AX, BX

The machine code is formed as shown by referring to data sheet Fig. 2.4 and using the Tables 2.1 and 2.2 as has been already described.

OPCODE	D	W	MOD	REG	R/M
0 0 0 0 0 0	1	1	11	0 0 0	0 1 1

The machine code is 03 C3.

**Example 3.12**

ADD AX, 5000H

The code formation is explained as follows:

OPCODE	S	W	MOD	OPCODE	R/M
1 0 0 0 0 0	0	1	0 0	0 0 0	0 0 0
8 1			0		0

LOWER BYTE DATA	HIGHER BYTE DATA
0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 0
0 0	5 0

The machine code is 81 00 00 50.

If S bit is 0, the 16-bit immediate data is available in the instruction.

If S bit is 1, the 16-bit immediate data is the sign extended form of 8-bit immediate data.

For example, if the eight bit data is 11010001, then its sign extended 16-bit version will be 11111111 11010001.

**Example 3.13**

SHR AX

OPCODE	V	W	MOD	REG	R/M
1 1 0 1 0 0	0	1	1 1	1 0 1	0 0 0
D 1			E	8	

The instruction code is D1 E8.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A program called 'assembler' is used to convert the mnemonics of instructions alongwith the data into their equivalent object code modules. These object code modules may further be converted in executable code using the linker and loader programs. This type of programming is called assembly level programming. In assembly language programming, the mnemonics are directly used in the user programs. The assembler performs the task of coding. The advantages of assembly language over machine language are as given:

1. The programming in assembly language is not so complicated as in machine language because the function of coding is performed by an assembler.
2. The chances of error being committed are less because the mnemonics are used instead of numerical opcodes. It is easier to enter an assembly language program.
3. As the mnemonics are purpose-suggestive the debugging is easier.
4. The constants and address locations can be labeled with suggestive labels hence imparting a more friendly interface to user. Advanced assemblers provide facilities like macros, lists, etc. making the task of programming much easier.
5. The memory control is in the hands of users as in machine language.
6. The results may be stored in a more user-friendly form.
7. The flexibility of programming is more in assembly language programming as compared to machine language because of advanced facilities available with the modern assemblers.

Basically, the assembler is a program that converts an assembly input file also called as source file to an object file that can further be converted into machine codes or an executable file using a linker. The recent versions of the assembler are designed with many facilities like macroassemblers, numerical processor assemblers, procedures, functions and so on. A discussion on the principles of assembler design and its working is presented in Chapter 12.

As far as this book is concerned, we will consider the assembly language programming using MASM (Microsoft Macro Assembler). There are a number of assemblers available like MASM, TASM and DOS assembler. MASM is one of the popular assemblers used along with a LINK program to structure the codes generated by MASM in the form of an executable file. MASM reads the source program as its input and provides an object file. The LINK accepts the object file produced by MASM as input and produces an EXE file.

While writing a program for an assembler, your first step will be to use a text editor and type the program listing prepared by you. Then check the listing typed by you for any typing mistake and syntax error. Before you quit the editor program, do not forget to save it. Once you save the text file with any name (permissible on operating system), you are free to start the assembly process. A number of text editors are available in the market, e.g. Norton's Editor [NE], Turbo C [TC], EDLIN, etc. Throughout this book, the NE is used. Any other free form editor may be used for a better user-friendly environment. Thus for writing a program in assembly language, one will need NE editor, MASM assembler, linker and DEBUG utility of DOS. In the following section, the procedures of opening a file for a program, assembling it, executing it and checking its result are described for beginners.

### 3.3.1 Entering a Program

In this section, we will explain the procedure for entering a small program on IBM PC with DOS operating system. Consider a program of addition of two bytes, as already discussed in the Section 3.1



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Once the above procedure is completed, you may now focus on assembling the program. Note that all the commands and displays shown in this section are for Norton's Editor. Other editors may require some other commands and their display style may be somewhat different but the overall procedure is the same.

ASSUME DATA	CS:CODE, DS:DATA SEGMENT OPR1 DW 1234 H OPR2 DW 0002 H RESULT DW 01 H DUP(?)
DATA CODE START	ENDS SEGMENT MOV AX, DATA MOV DS, AX MOV AX, OPR 1 MOV BX, OPR 2 CLC ADD AX, BX MOV DI, OFFSET RESULT MOV [DI], AX MOV AH, 4CH INT 21 H
CODE	ENDS END START
KBM.ASM	

**Fig. 3.9 A Program KMB.ASM in Norton's Editor**

Note that before quitting the editor program the newly entered or modified file must be saved, otherwise it will be lost and will not be available for further assembling process.

### 3.3.2 Assembling a Program

Microsoft Assembler MASM is an easy to use and popular assemblers. This section deals with the MASM. As already discussed, the main task of any assembler program is to accept the `text_assembly` language program file as an input and prepare an object file. The `text_assembly` language program file is prepared by using any of the editor programs as discussed in Section 3.3.1. The MASM accepts the file names only with the extension `.ASM`. Even if a filename without any extension is given as input, it provides an `.ASM` extension to it. For example, to assemble the program in Fig. 3.9, one may enter the following command options:

C> MASM KMB

or

C> MASM KMB.ASM

If any of the command option is entered as above, the screen displays, as shown in Fig. 3.10.

```
C>MASM KMB  
Microsoft @ Macro Assembler Version 5.10  
Copyright (c) Microsoft Corp.1981, 1989.  
All Rights Reserved  
  
Object filename[.OBJ] :  
List filename[NUL.LST] :  
Cross Reference[NUL.CRF] :
```

**Fig. 3.10 MASM Screen Display**

Another command option, available in MASM, that does not need any filename in the command line, is given along with the corresponding result display in Fig. 3.11.

```
C>MASM  
Microsoft @ Macro Assembler Version 5.10  
Copyright (c) Microfoft Corp.1981, 1989.  
All Rights Reserved  
  
Source filename[.ASM] :  
Object filename[FILE.OBJ] :  
List filename[NUL.LST] :  
List filename[NUL.CRF] :
```

**Fig. 3.11 MASM Alternative Screen Display**

If you do not enter the filename to be assembled at the command line as shown in Fig. 3.10, then you may enter it as a source filename as shown in Fig. 3.11. The source filename is to be typed in the source filename line with or without the extension .ASM. The valid filename entry is accepted with a pressure of enter key. On the next line, the expected .OBJ filename is to be entered which creates the object file of the assembly language program. The .OBJ file is created with the entered name and the .OBJ extension. If no filename is entered for it before pressing enter key, the new .OBJ file is created with the same name as source file and extension .OBJ.

The .OBJ file contains the coded object modules of the program to be assembled. On the next line, a filename is entered for the expected listing file of the source file, in the same way as the object filename was entered. The listing file is automatically generated in the assembly process. The listing file is identified by the entered or source filename and an extension .LST. This file contains the total offset map of the source file including labels, offset addresses, opcodes, memory allotment for different

labels and directives and relocation information. The cross reference filename is also entered in the same way as discussed for the listing file. This file is used for debugging the source program. It contains the statistical information size of the file in bytes, number of labels, list of labels, routines to be called, etc. about the source program.

After the cross-reference file name is entered, the assembly process starts. If the program contains syntax errors, they are displayed using error code number and the corresponding line number at which they appear. Once these syntax errors and warnings are taken care of by the programmer, the assembly process is completed successfully. The successful assembly process may generate the .OBJ, .LST and .CRF files, which may further be used by the linker programmer to link the object modules and generate an executable (.EXE) file from a .OBJ file. All the above said files may not be generated during the assembling of every program. The generation of some of them may be suppressed using the specific command line options of MASM. The discussions regarding the different command line options of MASM are out of the scope of this book. Here we intend to highlight just a routine assembling procedure. The files generated by the MASM are further used by the program LINK.EXE to generate an executable file of the source program.

### 3.3.3 Linking a Program

The DOS linking program LINK.EXE links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by MASM. The linker program is invoked using the following options.

C> LINK

or

C> LINK KMB.OBJ

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first option may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The other option also generates the similar display, but will not ask for the .OBJ filename, as it is already specified at the command line. If no filenames are entered for these files, by default, the source filename is considered with different extensions. The procedure of entering the filenames in LINK is also similar to that in MASM. The LINK command display is as shown in Fig. 3.12.

```
C>LINK
Microsoft @ Overlay Linker Version. 3.64
Copyright(c) Microsoft Corp. 1983-88. All Rights Reserved.

Object Module[.OBJ]:
Run file[.EXE]:
List file[NUL.MAP]:
Libraries[LIB]:
```

Fig. 3.12 Link Command Screen Display



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In this chapter, a few example programs are presented. Starting from very simple programs, the chapter concludes with more complex programs.

### Program 3.1

Write a program for addition of two numbers.

**Solution** The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes that the programmer wants to use. Accordingly, there may be different program listings to achieve a single programming goal. A skilled programmer uses a simple logic and implements it by using a minimum number of instructions. Let us now try to explain the following program:

```
ASSUME CS:CODE,DS:DATA

DATA SEGMENT
OPR1 DW 1234H ; 1st operand
OPR2 DW 0002H ; 2nd operand
RESULT DW 01 DUP(?) ; A word of memory reserved for result
DATA ENDS
CODE SEGMENT
START: MOV AX,DATA ; Initialize data segment
       MOV DS,AX ;
       MOV AX,OPR1 ; Take 1st operand in AX
       MOV BX,OPR2 ; Take 2nd operand in BX
       CLC ; Clear previous carry if any
       ADD AX,BX ; Add BX to AX
       MOV DI,OFFSET RESULT ; Take offset of RESULT in DI
       MOV [DI],AX ; Store the result at memory address in DI
       MOV AH,4CH ; Return to DOS prompt
       INT 21H
CODE ENDS ; CODE segment ends
END START ; Program ends
```

**Program 3.1(a) Listings**

#### 3.4.1 How to Write an Assembly Language Program

The first step in writing an assembly language program is to define and study the problem. Then, decide the logical modules required for the program. From the statement of the program one may guess that some data may be required for the program or the result of the program that is to be stored in memory. Hence the program will need a logical space called DATA segment. Invariably the CODE segment is a part of a program containing the actual instruction sequence to be executed. If the stack facility is to be used in the program, it will require the STACK segment. The EXTRA segment may be used as an additional destination data segment. Note that the use of all these logical segments is not compulsory except for the CODE segment. Some programs may require DATA and CODE segments, while the others may also contain STACK and EXTRA. For example, Program 3.1 (a) requires only DATA and CODE segment.

The first line of the program containing the 'ASSUME' directive declares that the label CODE is to be used as a logical name for CODE segment and the label DATA is to be used for DATA segment. These labels CODE and DATA are reserved by MASM for these purposes only. They should not be

used as general labels. Once this statement is written in the program, CODE refers to the code segment and DATA refers to data segment throughout the program. If you want to change it in a program you will have to write another ASSUME statement in the program.

The second statement, DATA SEGMENT marks the starting of a logical data space DATA. Inside the DATA segment, OPR1 is the first operand. The directive DW defines OPR1 as a word operand of value 1234H and OPR2 as a word operand of value 0002H. The third DW directive reserves 01H words of memory for storing the result of the program and leaves it undefined due to the directive DUP(?). The statement DATA ENDS marks the end of the DATA segment. Thus the logical space DATA contains OPR1, OPR2 and RESULT, which will be allotted physical memory locations whenever the logical SEGMENT DATA is allocated memory or loaded in the memory of a computer as explained in the previous topic of relocation. The assembler calculates that the above data segment requires 6 bytes, i.e. 2 bytes each for OPR1, OPR2 and RESULT.

The code segment in the above program starts with the statement CODE SEGMENT. The code segment, as already explained, is a logical segment space containing the instructions. The label STARTS marks the starting point of the execution sequence. The ASSUME statement just informs the assembler that the label CODE is used for the code segment and the label DATA is used for the DATA segment. It does not actually put the address corresponding to CODE in Code Segment (CS) register and address corresponding to DATA in the Data Segment (DS) register. This procedure of putting the actual segment address values into the corresponding segment registers is known as segment register initialisation. A programmer has to carry out these initializations for DS, SS and ES using instructions, while the CS is automatically initialised by the loader at the time of loading the EXE file into the memory for actual execution. The first two instructions in the program are for data segment initialization.

Note that, no segment register in 8086 can be loaded with immediate segment address value, instead the address value should be first loaded into any one of the general purpose registers which can then be transferred to any of the segment registers DS, ES and SS. Also one should note that CS cannot be loaded at all. Its contents can be changed by using a long jump instruction, a call instruction or an interrupt instruction. For each of the segments DS, ES and SS, the programmer will have to carry out initialization if they are used in the program, while CS is automatically initialized by the loader program at the time of loading and execution. Then the two instructions move the two operands OPR1 and OPR2 in AX and BX respectively. Carry is cleared before addition operation (optional in this program). The ADD instruction will add BX into AX and store the result in AX. The instruction used to store the result in RESULT uses a different addressing mode than that used for taking OPR1 into AX. The indexed addressing mode is used to store the result of addition in memory locations labeled RESULT.

The instruction MOV DI, OFFSET RESULT stores the offset of the label RESULT into DI register. The next instruction stores the result available in AX into the address pointed to by DI, i.e. address of the RESULT. A lot has been already discussed about the function calls under INT 21H. The function value 4CH is for returning to the DOS prompt. If instead of these one writes HLT instruction there will not be any difference in program execution except that the computer will hang as the processor goes to HLT state, and the user will not be able to examine the result. In that case, for further operation, one will have to reset the computer and boot it again. To avoid this resetting of the computer every time you run the program and enable it to check the result, it is better to use the function call 4CH at the end of each program so that after executing the program, the computer returns back to DOS prompt. The

statement CODE ENDS marks the end of the CODE segment. The statement END START marks the end of the procedure that started with the label START. At the end of each file, the END statement is a must.

Until now, we have discussed Program 3.1(a) in significant detail. As we have already said, the program contains two logical segments CODE and DATA, but it is not at all necessary that all the programs must contain the two segments. A programmer may use a single segment to cover up data as well as instructions. Program 3.1(b) explains the fact.

---

```

ASSUME CS:CODE
        CODE      SEGMENT
        OPR1      DW      1234H
        OPR2      DW      0002H
        RESULT    DW      01 DUP(?)
START :   MOV AX, CODE
          MOV DS, AX
          MOV AX, OPR1
          MOV BX, OPR2
          CLC
          ADD AX, BX
          MOV DI, OFFSET RESULT
          MOV [DI], AX
          MOV AH, 4CH
          INT 21H
CODE     ENDS
END START

```

---

#### **Program 3.1(b)** Alternative listing for Program 3.1

We have discussed all the properties of this program in detail. For all the following programs, we will not explain the common things like forming segments using directives and operators, etc. Instead, just the logic of the program will be explained. The use of proper syntax of the 8086/8088 assembler MASM is self explanatory. The comments may help the reader in getting the ideas regarding the logic of the program.

Assemble the above written program using MASM after entering it into the computer using the procedure explained in Section 3.3.1. Once you get the EXE file as the output of the LINK program, Your listing is ready for execution. The Program 3.1 is prepared in the form of EXE file with the name KMB.EXE in the directory. Next, it can be executed with the command line as given below.

C> KMB

This method of execution will store the result of the program in memory but will not display it on the screen. To display the result on the screen the programmer will have to use DOS function calls, which will make the programs too lengthy. Hence, another method to check the results is to run the program under the control of DEBUG. To run the program under the control of debug and to observe the results one must prepare the LST file, that gives information about memory allotment to different labels and variables of the program while assembling it. The LST file can be displayed on the screen using NE-Norton's Editor.

**Program 3.2**

Write a program for the addition of a series of 8-bit numbers. The series contains 100(numbers).

**Solution** In the first program, we have implemented the addition of two numbers. In this program, we show the addition of 100 (D) numbers. Initially, the resulting sum of the first two numbers will be stored. To this sum, the third number will be added. This procedure will be repeated till all the numbers in the series are added. A conditional jump instruction will be used to implement the counter checking logic. The comments explain the purpose of each instruction.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
NUMLIST DB 52H,23H,-
COUNT EQU 100D
RESULT DW 01H DUP(?)
DATA ENDS
CODE SEGMENT
ORG 200H
START:    MOV AX,DATA           ; Data segment starts
          MOV DS,AX            ; List of byte numbers
          MOV CX,COUNT          ; Number of bytes to be added
          XOR AX,AX             ; One word is reserved for result
          DATA ENDS             ; Data segment ends
          CODE SEGMENT          ; Code segment starts at relative
          ORG 200H              ; address 0200h in code segment
          START:                ; Initialize data segment
          MOV AX,DATA
          MOV DS,AX
          MOV CX,COUNT
          XOR AX,AX
          XOR BX,BX
          MOV SI,OFFSET NUMLIST
          AGAIN:    MOV BL,[SI]           ; Number of bytes to be added in CX
          ADD AX,BX             ; Clear AX and CF
          INC SI                ; Clear BH for converting the byte to word
          DEC CX                ; Point to the first number in the list
          JNZ AGAIN              ; Take the first number in BL,BH is zero
          ADD AX,BX             ; Add AX with BX
          INC SI                ; Increment pointer to the byte list
          DEC CX                ; Decrement counter
          JNZ AGAIN              ; If all numbers are added, point to result
          MOV DI,OFFSET RESULT
          MOV [DI],AX            ; destination and store it
          MOV AH,4CH              ; Return to DOS
          INT 21H
          CODE ENDS
END       START

```

**Program 3.2 Listings**

The use of statement ORG 200H in this program is not compulsory. We have used this statement here just to explain the way to use it. It will not affect the result at all. Whenever the program is loaded into the memory whatever is the address assigned for CODE, the executable code starts at the offset address 0200H due to the above statement. Similar to DW, the directive DB reserves space for the list of 8-bit numbers in the series. The procedure for entering the program, coding and execution has already been explained. The result of addition will be stored in the memory locations allotted to the label RESULT.

**Program 3.3**

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the locations starting from a known address.

**Solution** Compare the  $i$ th number of the series with the  $(i+1)$ th number using CMP instruction. It will set the flags appropriately, depending upon whether the  $i$ th number or the  $(i+1)$ th number is greater. If the  $i$ th number is greater than  $(i+1)$ th, leave it in AX (any register may be used). Otherwise, load the  $(i+1)$ th number in AX, replacing the  $i$ th number in AX. The procedure is repeated till all the members in the array have been compared.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DB 52H,23H,56H,45H,--
COUNT EQU OF
LARGEST DB 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:    MOV AX,DATA
          MOV DS,AX
          MOV SI,OFFSET LIST
          MOV CL,COUNT
          MOV AL,[SI]
AGAIN:   CMP AL,[SI+1]
          JNL NEXT
          MOV AL,[SI+1]
NEXT:    INC SI
          DEC CL
          JNZ AGAIN
          MOV SI,OFFSET LARGEST
          MOV [SI],AL
          MOV AH,4CH
          INT 21H
          CODE ENDS
END      START

```

; Data segment starts  
; List of byte numbers  
; Number of bytes in the list  
; One byte is reserved for the largest number.  
; Data segment ends  
; Code segment starts.  
; Initialize data segment.

; Number of bytes in CL.  
; Take the first number in AL  
; and compare it with the next number.

; Increment pointer to the byte list.  
; Decrement counter.  
; If all numbers are compared, point to result  
; destination and store it.

; Return to DOS.

### Program 3.3 Listings

#### Program 3.4

Modify the Program 3.3 for a series of words.

**Solution** The logic is similar to the previous program written for a series of byte numbers. The program is directly written as follows without any comment leaving it to the reader to find out the use of each instruction and directive used.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 1234H,2354H,0056H,045AH,-
COUNT EQU OF
LARGEST DW 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:    MOV AX,DATA

```

```

        MOV DS,AX
        MOV SI,OFFSET LIST
        MOV CL,COUNT
        MOV AX,[SI]
AGAIN:   CMP AX,[SI+2]
        JNL NEXT
        MOV AX,[SI+2]
NEXT:    INC SI
        INC SI
        DEC CL
        JNZ AGAIN
        MOV SI,OFFSET LARGEST
        MOV [SI],AX
        MOV AH,4CH
        INT 21H
CODE     ENDS
END      START

```

**Program 3.4 Listings****Program 3.5**

A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

**Solution** The simplest logic to decide whether a binary number is even or odd, is to check the least significant bit of the number. If the bit is zero, the number is even, otherwise it is odd. Check the LSB by rotating the number through carry flag, and increment even or odd number counter.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 2357H,0A579H,0C322H,0C91EH,0C000H,0957H
COUNT EQU 006H
DATA ENDS
CODE SEGMENT
START:    XOR BX,BX
          XOR DX,DX
          MOV AX,DATA
          MOV DS,AX
          MOV CL,COUNT
          MOV SI, OFFSET LIST
AGAIN:    MOV AX,[SI]
          ROR AX,01
          JC ODD
          INC BX
          JMP NEXT
ODD:      INC DX
NEXT:    ADD SI,02

```

```
DEC CL  
JNZ AGAIN  
MOV AH,4CH  
INT 21H  
CODE ENDS  
END START
```

---

**Program 3.5 Listings**

---

---

**Program 3.6**

Write a program to find out the number of positive numbers and negative numbers from a given series of signed numbers.

**Solution** Take the *i*th number in any of the registers. Rotate it left through carry. The status of carry flag, i.e. the most significant bit of the number will give the information about the sign of the number. If CF is 1, the number is negative; otherwise, it is positive.

```
ASSUME CS:CODE,DS:DATA  
DATA SEGMENT  
LIST DW 2579H,0A500H,0C009H,0159H,0B900H  
COUNT EQU 05H  
DATA ENDS  
CODE SEGMENT  
START:    XOR BX,BX  
          XOR DX,DX  
          MOV AX,DATA  
          MOV DS,AX  
          MOV CL,COUNT  
          MOV SI,OFFSET LIST  
AGAIN:    MOV AX,[SI]  
          SHL AX,01  
          JC NEG  
          INC BX  
          JMP NEXT  
NEG:      INC DX  
NEXT:     ADD SI,02  
          DEC CL  
          JNZ AGAIN  
          MOV AH,4CH  
          INT 21H  
          CODE ENDS  
          END START
```

---

**Program 3.6 Listings**

---

The logic of Program 3.6 is similar to that of Program 3.5, hence comments are not given in Program 3.6 except for a few important ones.

**Program 3.7**

Write a program to move a string of data words from offset 2000H to offset 3000H the length of the string is 0FH.

**Solution** To write this program, we will use an important facility, available in the 8086 instruction set, i.e. move string byte/word instruction. We will also study the flexibility imparted by this instructions to the 8086 assembly language program. Let us first write the Program 3.7 for 8085, assuming that the string is available at location 2000H and is to be moved at 3000H.

```

LXI H , 2000H
LXI D , 3000H
MVI C , 0FH
AGAIN :    MOV A , M
            STAX D
            INX H
            INX D
            DCR C
            JNZ AGAIN
            HLT

```

*An 8085 Program for Program 3.7*

Now assuming DS is suitably set, let us write the sequence for 8086. At first using the index registers, the program can be written as given:

```

MOV SI ,2000H
MOV DI ,3000H
MOV CX ,0FH
AGAIN :   MOV AX ,[SI]
           MOV [DI], AX
           ADD SI, 02H
           ADD DI, 02H
           DEC CX
           JNZ AGAIN
           HLT

```

*An 8086 Program for Program 3.7*

Comparing the above listings for 8085 and 8086, we may infer that every instruction in 8085 listing is replaced by an equivalent instruction of 8086. The above 8086 listing is absolutely correct but it is not efficient. Let us try to write the listings for the same purpose using the string instruction. Due to the assembler directives and the syntax, one may feel that the program is lengthy, though it eliminates four instructions for a MOVSW instruction.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
SOURCESTRT EQU 2000H
DESTSTRT EQU 3000H
COUNT EQU OFH
DATA ENDS
CODE SEGMENT

```

```

START:      MOV AX,DATA
            MOV DS,AX
            MOV ES,AX
            MOV SI,SOURCESTR
            MOV DI,DESTSTR
            MOV CX,COUNT
            CLD
REP        MOVSW
            MOV AH,4CH
            INT 21H
CODE ENDS
END START

```

**Program 3.7** An 8086 Program listing for Program 3.7 using String instruction

Compare the above two 8086 listings. Both contain ten instructions. However, in case of the second program, the instruction for the initialisation of segment register and DOS interrupt are additional while the first one neither contains initialisation of any segment registers nor does it contain the DOS interrupt instruction. We can say that the first program uses 9 instructions, while the second one uses only 5 for implementing the same algorithm.

This program and the related discussions are aimed at explaining the importance of the string instructions and the method to use them.

### Program 3.8

Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

**Solution** There exist a large number of sorting algorithms. The algorithm used here is called *bubble sorting*. The method of sorting is explained as follows. To start with, the first number of the series is compared with the second one. If the first number is greater than second, exchange their positions in the series otherwise leave the positions unchanged. Then, compare the second number in the recent form of the series with third and repeat the exchange part that you have carried out for the first and the second number, and for all the remaining numbers of the series. Repeat this procedure for the complete series ( $n-1$ ) times. After ( $n-1$ ) iterations, you will get the largest number at the end of the series, where  $n$  is the length of the series. Again start from the first address of the series. Repeat the same procedure right from the first element to the last element. After ( $n-2$ ) iterations you will get the second highest number at the last but one place in the series. Continue this till the complete series is arranged in ascending order. Let the series be as given:

53 , 25 , 19, 02	$n = 4$
25 , 53 , 19, 02	1st operation
25 , 19 , 53 , 02	2nd operation
25 , 19 , 02 , 53	3rd operation
largest no. $\Rightarrow$	$4 - 1 = 3$ operations
19 , 25 , 02 , 53	1st operation
19 , 02 , 25 , 53	2nd operation
2nd largest number $\Rightarrow$	$4 - 2 = 2$ operations
02 , 19 , 25 , 53	1st operation
3rd largest number $\Rightarrow$	$4 - 3 = 1$ operations

Instead of taking a variable count for the external loop in the program like  $(n - 1)$ ,  $(n - 2)$ ,  $(n - 3)$ , ..., etc. It is better to take the count  $(n - 1)$  all the time for simplicity. The resulting program is given as shown.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DW 53H,25H,19H,02H
COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:    MOV AX,DATA
          MOV DS,AX
          MOV DX,COUNT-1
AGAINO:   MOV CX,DX
          MOV SI,OFFSET LIST
AGAIN1:   MOV AX,[SI]
          CMP AX,[SI+2]
          JL PR1
          XCHG [SI+2],AX
          XCHG [SI],AX
PR1:      ADD SI,02
          LOOP AGAIN1
          DEC DX
          JNZ AGAINO
          MOV AH,4CH
          INT 21H
CODE      ENDS
END START

```

### Program 3.8 Listings

With a similar approach, the reader may write a program to arrange the string in descending order. For this, instead of the JL instruction in the above program, one will have to use a JG instruction.

### Program 3.9

Write a program to perform a one byte BCD addition.

**Solution** It is assumed that the operands are in BCD form, but the CPU considers it hexadecimal and accordingly performs addition. Consider the following example for addition. Carry is set to be zero.

$$\begin{array}{r}
 92 \\
 + 59 \\
 \hline
 \text{EB}
 \end{array} \quad \text{Actual result after addition considering hex. operands}$$

$$\begin{array}{r}
 1011 \\
 + 0110 \\
 \hline
 10001
 \end{array} \quad \begin{array}{l}
 \text{As } 0BH \text{ (LSD of addition)} > 09, \text{ add } 06 \text{ to it.} \\
 \text{Least significant nibble of result (neglect the auxiliary carry) } \rightarrow \text{AF is set to 1}
 \end{array}$$

0110 is added to most significant nibble of the result if it is greater than 9 or AF is set.

$$\begin{array}{r}
 & \quad \quad \quad 1 \quad \text{Carry from previous digit (AF)} \\
 E & \rightarrow 1 \ 1 \ 1 \ 0 \\
 + & 0 \ 1 \ 1 \ 0 \\
 \hline
 \text{CF is set to 1} & 0 \ 1 \ 0 \ 1 \quad \text{next significant nibble of result}
 \end{array}$$

Result CF	Most significant	Least significant digit
1	5	1

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
    OPR1 EQU 92H
    OPR2 EQU 52H
RESULT DB 02 DUP(00)
DATA ENDS
CODE SEGMENT
START:    MOV AX, DATA
          MOV DS,AX
          MOV BL,OPR1
          XOR AL,AL
          MOV AL,OPR2
          ADD AL,BL
          DAA
          MOV RESULT,AL
          JNC MSBO
          INC [RESULT+1]
MSBO:     MOV AH,4CH
          INT 21H
CODE ENDS
END START

```

### Program 3.9 Listings

In this program, the instruction DAA is used after ADD. Similarly, DAS can be used after SUB instruction. The reader may try to write a program for BCD subtraction for practice.

### Program 3.10

Write a program that performs addition, subtraction, multiplication and division of the given operands. Perform BCD operation for addition and subtraction.

**Solution** Here we have directly given the routine for Program 3.10.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
    OPR1 EQU 98H
    OPR2 EQU 49H
    SUM DW 01 DUP(00)
    SUBT DW 01 DUP(00)
    PROD DW 01 DUP(00)

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

        MOV ES,AX
        MOV CX,COUNT
        MOV DI,OFFSET STRING
        MOV BL,00H
        MOV AL,BYTE1
SCAN1:   NOP
        SCASB
        JZ XXX
        INC BL
        LOOP SCAN1
XXX:     MOV AH,4CH
        INT 21H
        CODE ENDS
DATA SEGMENT
BYTE1 EQU 25H
COUNT EQU 06H
STRING DB 12H,13H,20H,20H,25H,21H
DATA ENDS
END START

```

**Program 3.11 Listings****Program 3.12**

Write a program to convert the BCD numbers 0 to 9 to their equivalent seven segment codes using the look-up table technique. Assume the codes [7-seg] are stored sequentially in CODELIST at the relative addresses from 0 to 9. The BCD number (CHAR) is taken in AL.

**Solution** Refer to the explanation of the XLAT instruction. The statement of the program itself gives the explanation about the logic of the program.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
        CODELIST DB 34,45,56,45,23,12,19,24,21,00
        CHAR EQU 05
        CODEC    DB 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:   MOV AX,DATA
        MOV DS,AX
        MOV BX,OFFSET CODELIST
        MOV AL,CHAR
        XLAT
        MOV BYTE PTR CODEC,AL
        MOV AH,4CH
        INT 21H
CODE     ENDS
END START

```

**Program 3.12 Listings**



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Solution** In the addition of two matrices, the corresponding elements are added to form the corresponding elements of the result matrix as shown:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} + a_{12} + b_{12} + a_{13} + b_{13} \\ a_{21} + b_{21} + a_{22} + b_{22} + a_{23} + b_{23} \\ a_{31} + b_{31} + a_{32} + b_{32} + a_{33} + b_{33} \end{bmatrix}$$

$$[A] + [B] = [A + B]$$

The matrix A is stored in the memory at an offset MAT1, as given:

$a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}$ , etc.

A total of  $3 \times 3 = 9$  additions are to be done. The assembly language program is written as shown:

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
    DIM EQU 09H
    MAT1 DB 01,02,03,04,05,06,07,08,09
    MAT2 DB 01,02,03,04,05,06,07,08,09
    RMAT3 DW 09H DUP(?)

DATA ENDS
CODE SEGMENT
START:    MOV AX,DATA
          MOV DS,AX
          MOV CX,DIM
          MOV SI,OFFSET MAT1
          MOV DI,OFFSET MAT2
          MOV BX,OFFSET RMAT3
NEXT:     XOR AX,AX
          MOV AL,[SI]
          ADD AL, [DI]
          MOV WORD PTR [BX],AX
          INC SI
          INC DI
          ADD BX,02
          LOOP NEXT
          MOV AH,4CH
          INT 21H
CODE      ENDS
END START
```

#### Program 3.14 Listings

#### Program 3.15

Write a program to find out the product of two matrices. Store the result in the third matrix. The matrices are specified as in the Program 3.14.

**Solution** The multiplication of matrices is carried out as shown:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{12}b_{12} + a_{22}b_{22} + a_{32}b_{32} & a_{13}b_{13} + a_{23}b_{23} + a_{33}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

The listings to carry out the above operation is given as shown:

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
ROCOL EQU 03H
MAT1 DB 05H,09H,0AH,03H,02H,07H,03H,00H,09H
MAT2 DB 09H,07H,02H,01H,0H,0DH,7H,06H,02H
PMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:    MOV AX,DATA
          MOV DS,AX
          MOV CH,ROCOL
          MOV BX,OFFSET PMAT3
          MOV SI,OFFSET MAT1
NEXTROW:   MOV DI,OFFSET MAT2
          MOV CL,ROCOL
NEXTCOL:   MOV DL,ROCOL
          MOV BP,0000H
          MOV AX,0000H
          SAHF
NEXT_ELE:  MOV AL,[SI]
          MUL BYTE PTR[DI]
          ADD BP,AX
          INC SI
          ADD DI,03
          DEC DL
          JNZ NEXT_ELE
          SUB DI,08
          SUB SI,03
          MOV [BX],BP
          ADD BX,02
          DEC CL
          JNZ NEXTCOL
          ADD SI,03
          DEC CH
          JNZ NEXTROW
          MOV AH,4CH
```

```
INT 21H  
CODE ENDS  
END START
```

---

**Program 3.15 Listings**

---

**Program 3.16**

Write a program to add two multibyte numbers and store the result as a third number. The numbers are stored in the form of the byte lists stored with the lowest byte first.

**Solution** This program is similar to the program written for the addition of two matrices except for the addition instruction.

```
ASSUME CS:CODE,DS:DATA  
DATA SEGMENT  
BYTES EQU 08H  
NUM1 DB 05,5AH,6CH,55H,66H,77H,34H,12H  
NUM2 DB 04,56H,04H,57H,32H,12H,19H,13H  
NUM3 DB 0AH DUP(00)  
DATA ENDS  
CODE SEGMENT  
START:    MOV AX,DATA  
          MOV DS,AX  
          MOV CX,BYTES  
          MOV SI,OFFSET NUM1  
          MOV DI,OFFSET NUM2  
          MOV BX,OFFSET NUM3  
          XOR AX,AX  
NEXTBYTE:  MOV AL,[SI]  
          ADC AL,[DI]  
          MOV BYTE PTR[BX],AL  
          INC SI  
          INC DI  
          INC BX  
          DEC CX  
          JNZ NEXTBYTE  
          JNC NCARRY  
          MOV BYTE PTR[BX],01  
NCARRY:   MOV AH,4CH  
          INT 21H  
CODE ENDS  
END START
```

---

**Program 3.16 Listings**

---

**Program 3.17**

Write a program to add more than two multibyte numbers. The numbers are specified in a single list byte-wise one after another.

**Solution** In this program, all the numbers are stored in a single list byte-wise. The least significant byte of the first number is stored first, then the next significant byte and so on. After the most significant byte of the first number, the least significant byte of the second number will be stored. The series thus will end with the most significant byte of the last number. Let each number be of 8 bytes and 10 numbers are to be added. The list will contain  $8 \times 10 = 80$  bytes. The result may have more than 8 bytes. Let us assume that the result requires 9 bytes to be stored. A separate string of 9 bytes is reserved for the result. The result is also stored in the same form as the numbers. The assembly language program for this problem is given as shown.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BYTES EQU 04
NUMBERS EQU 02
NUMBERLIST DB 55H,22H,0BCH,0FFH,76H,56H,0FFH,0FOH
RESULT DB BYTES+1 DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX,DATA
             MOV DS,AX
             XOR AX,AX
             MOV BL,BYTES
             MOV SI,OFFSET NUMBERLIST
             MOV DI,OFFSET RESULT
             MOV CL,BYTES
NEXTBYTE:    MOV CH,NUMBERS
NEXTNUM:     MOV AL,[SI]
             ADD SI,BYTES
             ADD [DI],AL
             JNC NOCARY
             INC BYTE PTR[DI+1]
NOCARY:      DEC CH
             JNZ NEXTNUM
             SUB SI,BYTES
             SUB SI,BYTES
             INC DI
             INC SI
             DEC BL
             JNZ NEXTBYTE
             MOV AH,4CH
             INT 21H
CODE ENDS
END START

```

---

### Program 3.17 Listings

---



---

### Program 3.18

Write a program to convert a 16 bit binary number into equivalent BCD number.

**Solution** The program to convert the binary number into equivalent BCD number is developed below :

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BIN EQU
RESULT DW (?)
DATA ENDS
CODE SEGMENT
START :    MOV AX, DATA      ; INITIALIZE DATA SEGMENT
            MOV DS, AX
            MOV BX,BIN
            MOV AX, 0      ; INITIALIZE TO 0
            MOV CX, 0      ; INITIALIZE TO 0
CONTINUE :   CMP BX, 0      ; COMPARISION FOR ZERO BINARY
             NUMBER.
             JZ ENDPORG    ; IF ZERO END THE PROGRAM
             DEC BX        ; DECREMENT BX BY 1
             MOV AL,CL
             ADD AL,1
             DAA
             MOV CL,AL
             MOV AL,CH
             ADC AL,00H
             DAA
             MOV CH,AL
             JMP CONTINUE
ENDPROG :   MOV RESULT, CX ; STORING RESULT IN DATA SEGMENT
             MOV AH,4CH
             INT 21H
             CODE ENDS
             END START

```

#### Program 3.18 Listings

#### Program 3.19

Write a program to convert a BCD number into an equivalent binary number.

**Solution** A program to convert a BCD number into binary equivalent number is developed below.

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BCD_NUM EQU 4576H
BIN_NUM DW (?)
DATA ENDS
CODE SEGMENT
START :    MOV AX, DATA      ; INITIALIZE DATA SEGMENT
            MOV DS, AX
            MOV BX,BCD_NUM ; BX IS NOW HAVING BCD NUMBER

```

```

CONTINUE :    MOV CX,0          ; INITIALIZATION
              CMP BX,0          ; COMPARISON TO CHECK BCD NUM IS ZERO
              JZ ENDPORG        ; IF ZERO END THE PROGRAM
              MOV AL,BL          ; 8 LSBs OF NUMBER IS TRANSFERRED TO AL
              SUB AL,1           ; SUBTRACT ONE FROM AL
              DAS                ; DECIMAL ADJUST AFTER SUBTRACTION
              MOV BL,AL          ; RESULT IS STORED IN BL
              MOV AL,BH          ; 8 MSB IS TRANSFERRED TO AL
              SBB AL,00H         ; SUBTRACTION WITH BORROW
              DAS                ; DECIMAL ADJUST AFTER SUBTRACTION
              MOV BH,AL          ; RESULT BACK IN BH REGISTER
              INC CX             ; INCREMENT CX BY 1
              JMP CONTINUE       ; JUMP TO CONTINUE

ENDPROG :     MOV BIN_NUM,CX   ; RESULT IS STORED IN DATA SEGMENT
              MOV AH,4CH          ; TERMINATION OF PROGRAM
              INT 21H             ; TERMINATION OF PROGRAM
              CODE ENDS
              END START

```

**Program 3.19 Listings****Program 3.20**

Write a program to convert an 8 bit binary number into equivalent gray code.

**Solution** A program to convert an 8 bit binary number into equivalent gray code is written below.

Binary	→	B <sub>7</sub>	B <sub>6</sub>	B <sub>5</sub>	B <sub>4</sub>	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>
Gray	→	G <sub>7</sub>	G <sub>6</sub>	G <sub>5</sub>	G <sub>4</sub>	G <sub>3</sub>	G <sub>2</sub>	G <sub>1</sub>	G <sub>0</sub>

$$\begin{aligned}
 G_7 &= B_7 \\
 G_i &= B_i \oplus B_{i+1} \\
 \text{Where } i &= 0 \text{ to } 6
 \end{aligned}$$

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
NUM EQU 34H
RESULT DB (?)
DATA ENDS
START :    MOV AX, DATA      ; INITIALIZE DATA SEGMENT
            MOV DS, AX
            MOV AL,NUM        ; NUMBER IS TRANSFERRED TO AL
            MOV BL, AL
            CLC                ; CLEAR CARRY FLAG
            RCR AL,1          ; ROTATE THROUGH CARRY THE CONTENT
                                ; OF AL
            XOR BL,.AL        ; XORING BL AND AL TO GET GRAY CODE
            MOV RESULT, BL    ; STORING RESULT IN DMS
            MOV AH,4CH
            INT 21H

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Appendix-B tabulates the different function values and their purposes under the INT 21 H interrupt. With the help of the information in Appendix-B and the instruction set of 8086, we are able to access the hardware resources of the system like CRT, keyboard, hard disk, floppy disk, memory, etc. Also, the software resources like directory structure, file allocation table, can be referred by using the information in Appendix-B. With the help of a few simple programs, we now explain, how to use the particular resource. It is assumed that the computer system is working under DOS.

---

### Program 3.22

Display the message "The study of microprocessors is interesting." on the CRT screen of a microcomputer.

**Solution** A program to display the string is given as follows:

```

ASSUME      CS :CODE,DS :DATA
DATA        SEGMENT
MESSAGE     DB 0DH,0AH,"      STUDY OF MICROPROCESSORS IS INTERESTING",
              0DH,0AH,"$"
              ;PREPARING STRING OF THE MESSAGE
DATA        ENDS
CODE        SEGMENT
START:      MOV AX,DATA      ;INITIALIZE DS
            MOV DS,AX
            MOV AH,09H      ;SET FUNCTION VALUE FOR DISPLAY
            MOV DX,OFFSET MESSAGE
            INT 21H         ;POINT TO MESSAGE AND RUN
            MOV AH,4CH      ;THE INTERRUPT
            INT 21H         ;RETURN TO DOS
            END START
CODE        ENDS
            ;STOP

```

### Program 3.22 Listings

---

The above listing starts with the usual statement ASSUME. In the data segment, the message is written in the form of a string of the message characters and cursor control characters. The characters 0AH and 0DH are the line feed and carriage feed characters. The "\$" is the string termination character. At the end of every message to be displayed the "\$" must be there. Otherwise, the computer loses the control, as it is unable to find the end of the string. The character 0DH brings the cursor to next line. The character 0AH brings cursor to the next position (column wise). In case of DB operator, the characters written in the statement in inverted double commas are built in the form of their respective ASCII codes in the allotted memory bytes for the string.

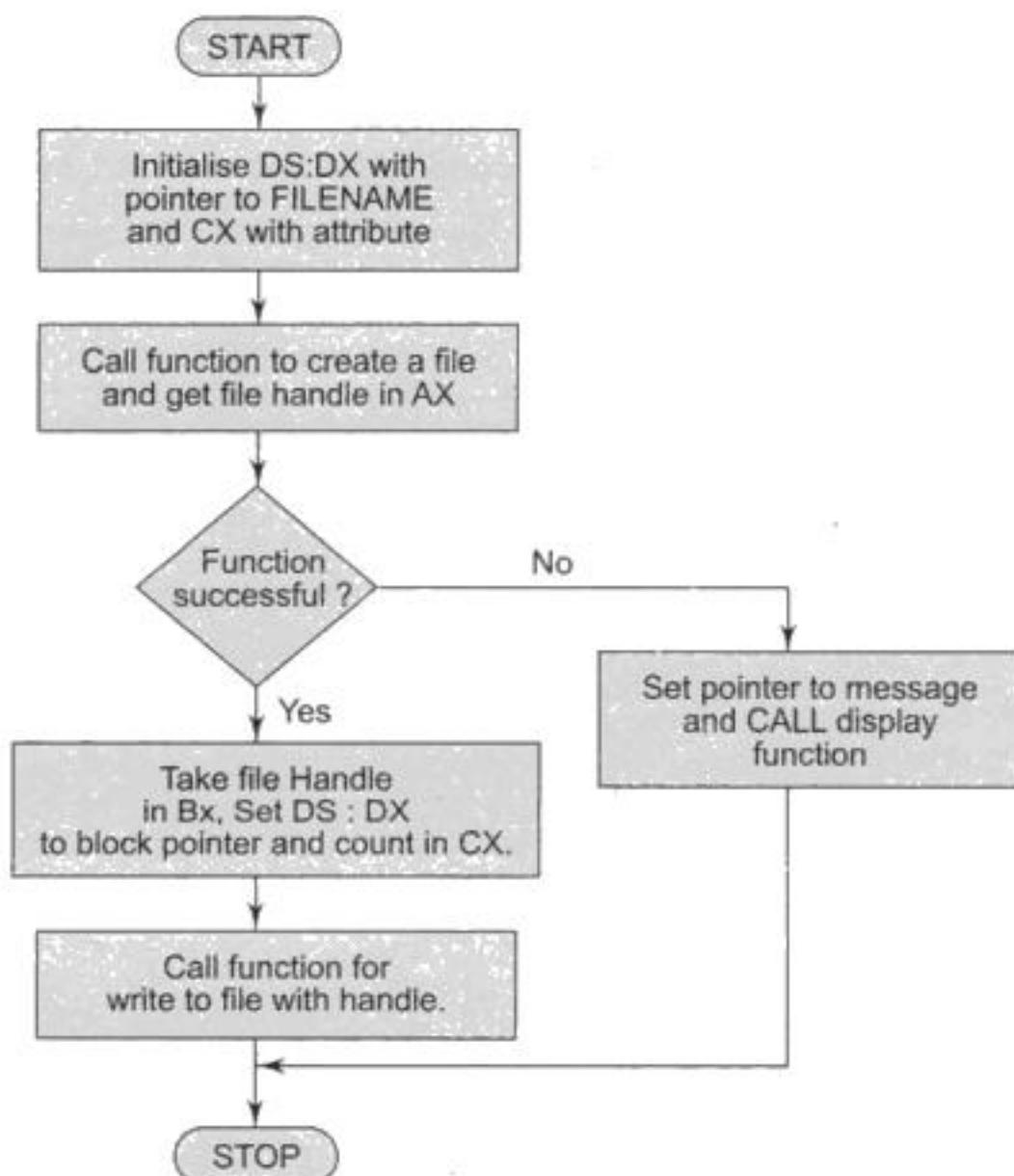
Then the data segment that contains the message string is initialised. The register AH is loaded with the function value 09H for displaying the message on the CRT screen. The instruction INT 21H after execution, causes the message to be displayed. The register DX points to the message in the data segment. After the message is displayed the control is returned to DOS prompt.

---

### Program 3.23

Write a program to open a new file KMB.DAT in the current directory and drive. If it is successfully opened, write 200H bytes of data into it from a data block named BLOCK. Display a message, if the file is not opened successfully.

**Solution** This type of programs, written for the utilization of resources of a computer system does not require much of logic. These programs contain the specific function calls along with some instructions to load the registers to prepare the environment (required data) for the interrupt call. A flow chart for Program 3.23 is shown in Fig. 3.14. It is up to the application designer to combine these programs with the actual application programs.



**Fig. 3.14 Flow Chart for Program 3.23**

```

ASSUME CS:CODE,DS:DATA
DATA SEGMENT
    DATABLOCK DB 200H DUP (?)
    FILENAME DB "KMB.DAT","$"
    MESSAGE DB 0AH,0DH,"FILE NOT CREATED
    SUCCESSFULLY",0AH,0DH,"$"
DATA ENDS
CODE SEGMENT
START: MOV AX,DATA ;INITIALIZE DS
        MOV DS,AX
        MOV DX,OFFSET FILENAME ;OFFSET OF FILE NAME
        MOV CX,00H ;FILE ATTRIBUTE IN CX,REFER TO
        MOV AH,3CH ;APPENDIX B FUNCTION 3CH
        INT 21H ;t0 CREATE A FILE, FOR more DE-
  
```

```

        ;TAILS.
JNC WRITE           ;IF file is CREATED SUCCESSFULLY
                    ;WRITE DATA IN TO IT

        MOV AX,DATA
        MOV DS,AX
        MOV DX,OFFSET MESSAGE
        MOV AH,09H
        INT 21H
        JMP STOP
WRITE:   MOV BX,AX
        MOV CX,0200H

        MOV DX,OFFSET DATABLOCK
        MOV AH,40H
        INT 21H
STOP:    MOV AH,4CH
        INT 21H
CODE     ENDS
END START

```

**Program 3.23 Listings****Program 3.24**

Write a program to load a file KMB.EXE in the memory at the CS value of 5000H with zero relocation factor. The file is just to be observed and not to be executed (overlay loading).

**Solution** This type of loading of a file is called as overlay loading. The function value 4BH in AH and 03 in AL serves the purpose. The program is given as shown. The reader may refer to Appendix-B for details of the function calls.

```

ASSUME      CS:CODE;DS:DATA
DATA        SEGMENT
LODPTR    DB 00,50H,00,00
MESSAGE    DB 0AH,0DH,"LOADING FAILURE",0AH,0DH,"$"
FILENAME   DB "PROG3-5.EXE","$"
DATA        ENDS
CODE        SEGMENT
START:    MOV AX,DATA
          MOV DS,AX
          MOV DX,OFFSET FILENAME      ;SET DS:DX TO STRING CONTAINING
          MOV BX,OFFSET LODPTR       ;FILE NAME
          MOV AX,SEG LODPTR         ;SET ES&BX TO POINT
          MOV ES,AX                  ;A BLOCK CONTAINING
          MOV AX,4B03H                ;CS & RELOCATION FACTOR
          INT 21H                   ;LOAD FUNCTION VALUE AND
          JNC OKAY                 ;IF LOADING IS NOT SUCCESSFULL,
          MOV AX,DATA
          MOV DS,AX

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

REP      CMPSB          ; Compare the string with the buffer
        JNZ SORRY       ; If string does not match with buffer
                      ; display 'Sorry!', otherwise return
                      ; to DOS prompt.

SORRY:   MOV DX,OFFSET MESSAGE ; Set offset pointer to display 'Sorry!'.
        MOV AH,09H        ; Set function value to 09H under
        INT 21H          ; DOS interrupt 21h.
        JMP WAIT         ; Wait for the next entries.

CODE    ENDS
        END START

```

**Program 3.25 Listings**

In these programs, setting the supporting parameters and the function values is of prime importance. One may go through Appendix-B and try to write more and more programs for different functions available under INT 21H of DOS.

**SUMMARY**

*This chapter starts with simple programs written for hand-coding. Further, hand-coding procedures of a few example instructions are studied. Advantages of the assembly language over machine language are then presented in brief. With an overview of the assembler operation, we have initiated the discussion of assembly language programming. The procedures of writing, entering and coding the assembly language programs are then discussed in brief. Further, DOS debugging program - DEBUG, a basic tool for trouble-shooting the assembly language programs, is discussed briefly with an emphasis on the most useful commands. Then we have presented various examples of 8086/8088 programs those highlight the actual use and the syntax of the instructions and directives. Finally, a few programs that enable the programmer to access the computer system resources using DOS function calls are discussed. We do not claim that each program presented here is the most efficient one, rather it just suggests a way to implement the algorithm asked in the problem. There may be a number of alternate algorithms and program listings to implement a logic but amongst them a programmer should choose one which requires minimum storage, execution time and complexity.*

**Exercises****3.1 Find out the machine code for following instructions.**

- |                        |                       |                       |
|------------------------|-----------------------|-----------------------|
| (i) ADC AX,BX          | (ii) OR AX,[0500H]    | (iii) AND CX,[SI]     |
| (iv) TEST AX,5555H     | (v) MUL [SI+5]        | (vi) NEG 50[BP]       |
| (vii) OUT DX,AX        | (viii) LES DI,[0700H] | (ix) LEA SI,[BX+500H] |
| (x) SHL [BX+2000],CL   | (xi) RET 0200H        | (xii) CALL 7000H      |
| (xiii) JMP 3000H:2000H | (xiv) CALL [5000H]    | (xv) DIV [5000H]      |

- 3.2 Describe the procedure for coding the intersegment and intrasegment jump and call instructions.
- 3.3 Enlist the advantages of assembly language programming over machine language.
- 3.4 What is an assembler?
- 3.5 What is a linker?
- 3.6 Explain various DEBUG commands for troubleshooting executable programs. (Refer to MSDOS Encyclopedia by Ray Duncan.)
- 3.7 What are the DOS function calls?
- 3.8 Write an ALP to convert a four digit hexadecimal number to decimal number.
- 3.9 Write an ALP to convert a four digit octal number to decimal number.
- 3.10 Write an ALP to find out ASCII codes of alphanumeric characters from a look up table.
- 3.11 Write an ALP to change an already available ascending order byte string to descending order.
- 3.12 Write an ALP to perform a 16-bit increment operation using 8-bit instructions.
- 3.13 Write an ALP to find out average of a given string of data bytes neglecting fractions.
- 3.14 Write an ALP to find out decimal addition of sixteen four digit decimal numbers.
- 3.15 Write an ALP to convert a four digit decimal number to its binary equivalent.
- 3.16 Write an ALP to convert a given sixteen bit binary number to its GRAY equivalent.
- 3.17 Write an ALP to find out transpose of a 3x3 matrix.
- 3.18 Write an ALP to find out cube of an 8-bit hexadecimal number.
- 3.19 Write an ALP to display message 'Happy Birthday!' on the screen after a key 'A' is pressed.
- 3.20 Write an ALP to open a file in the drive C of your hard disk , accept 256 byte entries each separated by comma and then store all these 256 bytes into the opened file. Issue suitable messages where-ever necessary.
- 3.21 Write an ALP to print a message 'The Printer Is Busy' on to a dot matrix printer.
- 3.22 Write an ALP to load a file from hard disk of your system into RAM system at segment address 5000H with zero relocation factor.
- 3.23 Write an ALP that goes on accepting the keyboard entries and displays them on line on the CRT display. The control escapes to DOS prompt if enter key is pressed.
- 3.24 Write an ALP to set interrupt vector of type 50H to an address of a routine ISR in segment CODE1.
- 3.25 Write an ALP to set and get the system time. Assume arbitrary time for setting the system time.
- 3.26 What is the function 4CH under INT 21H?



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

1. Sample the output of a signal conditioning block.
2. Send start of conversion signal to ADC.
3. Wait for delay time (conversion time) and sense end of conversion signal.
4. After the conversion is over read the ADC output.
5. Take number of samples using steps 1 to 4.
6. Find proportional, differential and integral error signals.
7. Derive control signal from result of step 6.
8. Apply the control signal to control the flow of energy to the heater.

The above procedure contains eight steps. Steps 1 to 4 are to be repeated for each sample. Suppose, for finding out the proportional, integral and differential error signal one decides to take say ten samples, then steps 6 to 8 shall be repeated after each group of ten samples.

The above program needs to control the operation of ADC. Also, it computes the proportional, integral and differential control laws and then serves digital control signal to the control element to decide the appropriate action. This program needs a number of general purpose registers as there may be a number of intermediate results which should be temporarily stored. The 8086 has only four 16-bit general purpose registers. Here, the stack provides a sequential mechanism to store the partial results.

Thus instead of writing a single big program for such application, one should split it into number of subtasks that constitute the complete application and then write separate routines for each subtask. After that, prepare a main program that calls the specific routines for the specific tasks. Suppose the processor is executing a main program that calls a subroutine. After executing the main program up to the CALL instruction, the control will be transferred to the subroutine address. Now, the microprocessor must know where the control is to be returned after the execution of the subroutine. A similar problem may arise while handling interrupts. This address of re-entry into the main program may be stored onto the stack. Also, the stack is useful for storing the register status of the processor at the time of calling a subroutine and getting it back at the time of returning, so that the registers or memory locations already used during the main program can be reused by the subroutine without any loss of data. The stack mechanism provides a temporary storage of data in these cases.

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. The stack is a block of memory locations which is accessed using the SP and SS registers. In other words, it is a top-down data structure whose elements are accessed using a pointer that is implemented using the SP and SS registers. As we go on storing the data words onto the stack, the pointer goes on decrementing and on the other hand, the pointer goes on incrementing as we go on retrieving the word data. For each such access, the stack pointer is decremented or incremented by two. The stack is required in case of CALL instructions. The data in the stack, may again be transferred back from stack to register, whenever required by the CPU. The process of storing the data in the stack is called 'pushing into' the stack and the reverse process of transferring the data back from the stack to the CPU register is known as 'popping off' the stack. The stack is essentially *Last-In-First-Out (LIFO)* data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first. For example, let us consider a stack of, say, five books lying randomly on the table which one wants to arrange in a stack. He will place one book out of the five in a position and mark it with 1, then the second book is placed above it and marked 2 and so on. Thus all the five books can be placed one above the other and marked with respective numbers. If one wants to refer to them at some later time, the upper most book on top of the stack marked 5, will be accessed first. If one



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Suppose, a main program is being executed by the processor. At some stage during the execution of the program, all the registers in the CPU may contain useful data. In case there is a subroutine CALL instruction at this stage, there is a possibility that all or some of the registers of the main program may be modified due to the execution of the subroutine. This may result in loss of useful data, which may be avoided by using the stack. At the start of the subroutine, all the registers' contents of the main program may be pushed onto the stack one by one. After each PUSH operation SP will be modified as already explained before. Thus all the registers can be copied to the stack. Now these registers may be used by the subroutine, since their original contents are saved onto the stack. At the end of the execution of the subroutine, all the registers can get back their original contents by popping the data from the stack. The sequence of popping is exactly the reverse of the pushing sequence. In other words, the register or memory location that is pushed into the stack at the end should be popped off first.

#### 4.2.1 Programming for Stack

As it has been discussed in Chapter 1, the memory bank of 8086/88 is organised according to segments. The 8086/8088 has four segment registers, namely, CS, DS, SS and ES. Out of these segment registers, SS, i.e. 'stack segment register' contains the segment value for stack while SP contains the offset for the stack-top. In a program the stack segment can be defined in a similar way as the data segment. The ASSUME directive directs the name of the stack segment to the assembler. The SS register and the SP register must be initialised in the program suitably. Program 4.1 explains the use of the stack segment.

##### Program 4.1

Write a program to calculate squares of BCD numbers 0 to 9 and store them sequentially from 2000H offset onwards in the current data segment. The numbers and their squares are in the BCD format. Write a subroutine for the calculation of the square of a number.

The procedure of computing the square of a number is to be repeated for all the numbers. A subroutine for calculating the square is written which is called repetitively from the main program. The 8086/88 does not have single instruction for calculation of the square of a number. Thus you may calculate the square of a number using ADD and DAA instructions. The result of the ADD instruction is in hexadecimal format and it should be converted to decimal form, before it is stored into the memory. Here, one may ask: why not to use the MUL instruction for calculating the squares of the number? A point to be noted here is that, the MUL instruction does not calculate the square of a decimal number and moreover, the DAA instruction is to be used only after the ADD or ADC instructions.

One of the advantages of the subroutine is that, a recurring sequence of instructions can be assigned with a procedure name, which may be called again and again whenever required, resulting in a comparatively smaller sequence of instructions.

After a subroutine is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction, i.e. the starting address of the subroutine. Then the subroutine is executed. It may be noted that there should be an equal number of PUSH and POP instructions in the subroutine that has to be executed so that the SP contents at the time of calling the subroutine must be equal to the contents of SP at the time of executing the RET instruction, at the end of the subroutine. Otherwise, the control that should be returned to the next instruction after the CALL instruction will not be returned back properly.

The assembly language listing for the above procedure is given as shown. Note that 8086 does not support any instruction available for direct BCD packed multiplication to calculate the square of numbers. Hence to calculate the squares, the multiplication is implemented as successive addition, and the DAA instruction is used after each addition operation to convert the result in decimal format.

```

ASSUME CS : CODE, DS : DATA, SS : STACK
DATA      SEGMENT
          ORG 2000H
SQUARES DB OFH DUP (?)
DATA      ENDS
STACK      SEGMENT
STAKDATA DB 100H DUP (?)           ; Reserve 256 bytes for stack
STACK      ENDS
CODE      SEGMENT
START:  MOV AX, DATA              ; Initialise data segment
        MOV DS, AX
        MOV AX, STACK              ; Initialise stack segment
        MOV SS, AX
        MOV SP, OFFSET STAKDATA    ; Initialise stack pointer
        MOV CL, 0AH                 ; Initialise counter for numbers
        MOV SI, OFFSET SQUARES     ; Set pointer to array of squares
        MOV AL, 00                  ; Start from 00
NEXTNUM : CALL SQUARE             ; Calculate square
        MOV BYTE PTR [SI], AH       ; Store square in the array
        INC AL                     ; Go for the next number
        INC SI                     ; Increment the array pointer
        DCR CL                     ; Decrement count
        JNZ NEXTNUM                ; Stop, if CL = 0, else, continue
        MOV AH, 4CH                 ; Return to DOS prompt
        INT 21H
PROCEDURE SQUARE NEAR
        MOV BH,AL
        MOV CH,AL
        XOR AL,AL
AGAIN : ADD AL,CH
        DAA
        DCR CH
        JNZ AGAIN
        MOV AH,AL
        MOV AL,BH
        RET
SQUARE    ENDP
CODE      ENDS
END START

```

**Program 4.2**

Write a program to change a sequence of sixteen 2-byte numbers from ascending to descending order. The numbers are stored in the data segment. Store the new series at addresses starting from 6000 H. Use the LIFO property of the stack.

```

ASSUME CS : CODE, DS : DATA, SS : DATA
DATA SEGMENT
LIST DW 10H
STACKDATA DB FFH DUP (?)  

ORG6000H
RESULT DW 10H
DATA ENDS COUNT EQU 10H
CODE SEGMENT
START: MOV AX, DATA ; Initialize data segment and
          MOV DS, AX ; stack segment
          MOV SS, AX
          MOV SP, OFFSET LIST ; Initialize stack pointer
          MOV CL, COUNT ; Initialize counter for word number
          MOV BX, OFFSET RESULT + COUNT ; Initialize BX at last address
                                         ; (stack) for destination series
NEXT: POP AX ; Get the first word from series
      MOV DX, SP ; Save source stack pointer
      MOV SP, BX ; Get destination stack pointer
      PUSH AX ; Save AX to stack
      MOV BX, SP ; Save destination stack pointer
      MOV SP, DX ; Get source stack pointer for
                   ; the next number
      DCR CL ; Decrement count
      JNZ NEXT ; If count is not zero, go to the next num
      MOV AH, 4CH ; Else, return to DOS
      INT 21H ; prompt
CODE ENDS
END START

```

**Program 4.2 Listing**

The above program may also be written using just simple data transfer instructions. Here we have used the stack to change the order of data words.

It is a common practice to push all the registers to the stack at the start of a subroutine and pop it at the end of the subroutine so that the original contents are retrieved. Thus during the subroutine execution, all the registers except SP and SS are free for use.

The stack mechanism is also used in case of interrupt service routines to store the instruction pointer and code segment of the return address. The maximum size of stack segment like a general segment is 64 K. The stack size for a particular program may be set using DB or DW directive as per the requirements, as shown in the above program.

### 4.3 INTERRUPTS AND INTERRUPT SERVICE ROUTINES

The dictionary meaning of the word 'interrupt' is to break the sequence of operation. While the CPU is executing a program, an 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called *Interrupt Service Routine* (ISR). After executing ISR, the control is transferred back again to the main program which was being executed at the time of interruption.

Suppose you are reading a novel and have completed up to page 100. At this instant, your younger brother distracts you. You will somehow mark the line and the page you are reading, so that you may be able to continue after you attend to him. Say you have marked page number 101. You will now go to his room to solve his problem. While you are helping him a friend of yours comes and asks you for a textbook. Now, there are two options in front of you. The first is to make the friend wait till you complete serving your brother, and thereafter you serve his request. In this, you are giving less priority to your friend. The second option is to ask your brother to wait; remember the solution of his problem at the intermediate state; serve the friend; and after the friend is served, continue with the solution that was in the intermediate state. In this case, it may be said that you have given higher priority to your friend. After serving both of them, again you may continue reading from page 101 of the novel. Here, first you are interrupted by your brother. While you are serving your brother, you are again interrupted by a friend. This type of sequence of appearance of interrupts is called nested interrupt, i.e. interrupt within interrupt.

Whenever a number of devices interrupt a CPU at a time, and if the processor is able to handle them properly, it is said to have *multiple interrupt processing capability*. For example, 8085 has five hardware interrupt pins and it is able to handle the interrupts simultaneously under the control of software. In case of 8086, there are two interrupt pins, viz. NMI and INTR. The NMI is a *nonmaskable* interrupt input pin which means that any interrupt request at NMI input cannot be masked or disabled by any means. The INTR interrupt, however, may be masked using the Interrupt Flag (IF). The INTR, further, is of 256 types. The INTR types may be from 00 to FFH(or 00 to 255). If more than one type of INTR interrupt occurs at a time, then an external chip called programmable interrupt controller is required to handle them. The same is the case for INTR interrupt input of 8085. Interrupt Service Routines (ISRs) are the programs to be executed by interrupting the main program execution of the CPU, after an interrupt request appears. After the execution of ISR, the main program continues its execution further from the point at which it was interrupted.

### 4.4 INTERRUPT CYCLE OF 8086/8088

Broadly, there are two types of interrupts. The first out of them is *external interrupt* and the second is *internal interrupt*. In external interrupt, an external device or a signal interrupts the processor from outside or, in other words, the interrupt is generated outside the processor, for example, a keyboard interrupt. The internal interrupt, on the other hand, is generated internally by the processor circuit, or by the execution of an interrupt instruction. The examples of this type are divide by zero interrupt, overflow interrupt, interrupts due to INT instructions, etc.

Suppose an external device interrupts the CPU at the interrupt pin, either NMI or INTR of the 8086, while the CPU is executing an instruction of a program. The CPU first completes the execution of the current instruction. The IP is then incremented to point to the next instruction. The CPU then acknowledges the requesting device on its INTA pin immediately if it is a NMI, TRAP or Divide by Zero interrupt. If it is an INT request, the CPU checks the IF flag. If the IF is set, the interrupt request is acknowledged using the INTA pin. If the IF is not set, the interrupt requests are ignored. Note that the responses to the NMI, TRAP and Divide by Zero interrupt requests are independent of the IF flag. After an interrupt is acknowledged, the CPU computes the vector address from the type of the interrupt that may be passed to the interrupt structure of the CPU internally (in case of software interrupts, NMI, TRAP and Divide by Zero interrupts) or externally, i.e. from an interrupt controller in case of external interrupts. (The contents of IP and CS are next pushed to the stack. The contents of IP and CS now point to the address of the next instruction of the main program from which the execution is to be continued after executing the ISR. The PSW is also pushed to the stack.) The Interrupt Flag (IF) is cleared. The TF is also cleared, after every response to the single step interrupt. The control is then transferred to the interrupt service routine for serving the interrupting device. The new address of ISR is found out from the interrupt vector table. The execution of the ISR starts. If further interrupts are to be responded to during the time the first interrupt is being serviced, the IF should again be set to 1 by the ISR of the first interrupt. If the interrupt flag is not set, the subsequent interrupt signals will not be acknowledged by the processor, till the current one is completed. The programmable interrupt controller is used for managing such multiple interrupts based on their priorities. At the end of ISR the last instruction should be IRET. When the CPU executes IRET, the contents of flags, IP and CS which were saved at the start by the CALL instruction are now retrieved to the respective registers. The execution continues onwards from this address, received by IP and CS.

We now discuss how the 8086/88 finds out the address of an ISR. Every external and internal interrupt is assigned with a type (N), that is either implicit (in case of NMI, TRAP and divide by zero) or specified in the instruction INT N (in case of internal interrupts). In case of external interrupts, the type is passed to the processor by an external hardware like programmable interrupt controller. In the zeroth segment of physical address space, i.e. CS = 0000, Intel has reserved 1,024 locations for storing the interrupt vector table. The 8086 supports a total of 256 types of the interrupts, i.e. from 00 to FFH. Each interrupt requires 4 bytes, i.e. two bytes each for IP and CS of its ISR. Thus a total of 1,024 bytes are required for 256 interrupt types, hence the interrupt vector table starts at location 0000:0000 and ends at 0000:03FFH. The interrupt vector table contains the IP and CS of all the interrupt types stored sequentially from address 0000:0000 to 0000:03FFH. The interrupt type N is multiplied by 4 and the hexadecimal multiplication obtained gives the offset address in the zeroeth code segment at which the IP and CS addresses of the interrupt service routine (ISR) are stored. The execution automatically starts from the new CS:IP.

Figure 4.4 shows the interrupt sequence of 8086/88 and Fig. 4.5 shows the structure of interrupt vector table.

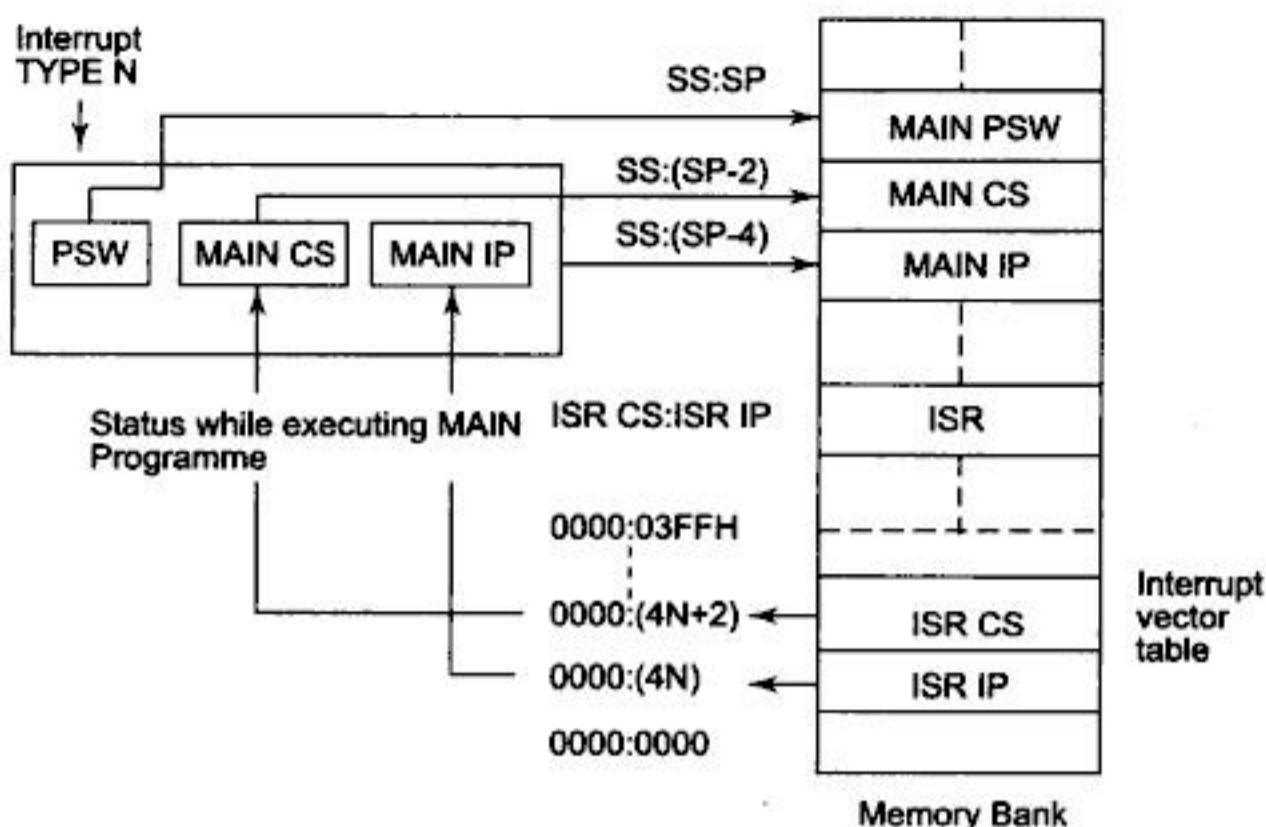


Fig. 4.4 Interrupt Response Sequence

Interrupt Type	Content (16-bit)	Address	Comments
Type 0	ISR IP ISR CS	0000:0000 0000:0002	Reserved for divide by Zero interrupt
Type 1	ISR IP ISR CS	0000:0004 0000:0006	Reserved for single step interrupt
Type 2	ISR IP ISR CS	0000:0008 0000:000A	Reserved for NMI
Type 3	ISR IP ISR CS	0000:000C 0000:000E	Reserved for INT single byte instruction
Type 4	ISR IP ISR CS	0000:0010 0000:0012	Reserved for INTO instruction
Type N	ISR IP ISR CS	0000:0014 0000:0016 0000:004N 0000:(004N+2)	Reserved for two byte instruction INT TYPE
Type FFH	ISR IP ISR CS	0000:03FC 0000:03FE 0000:03FF	

ISR : Interrupt Service Routine

Fig. 4.5 Structure of Interrupt Vector Table of 8086/88



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Suppose an external signal interrupts the processor and the pin **LOCK** goes low at the trailing edge of the first ALE pulse that appears after the interrupt signal preventing the use of bus for any other purpose.

The pin **LOCK** remains low till the start of the next machine cycle.

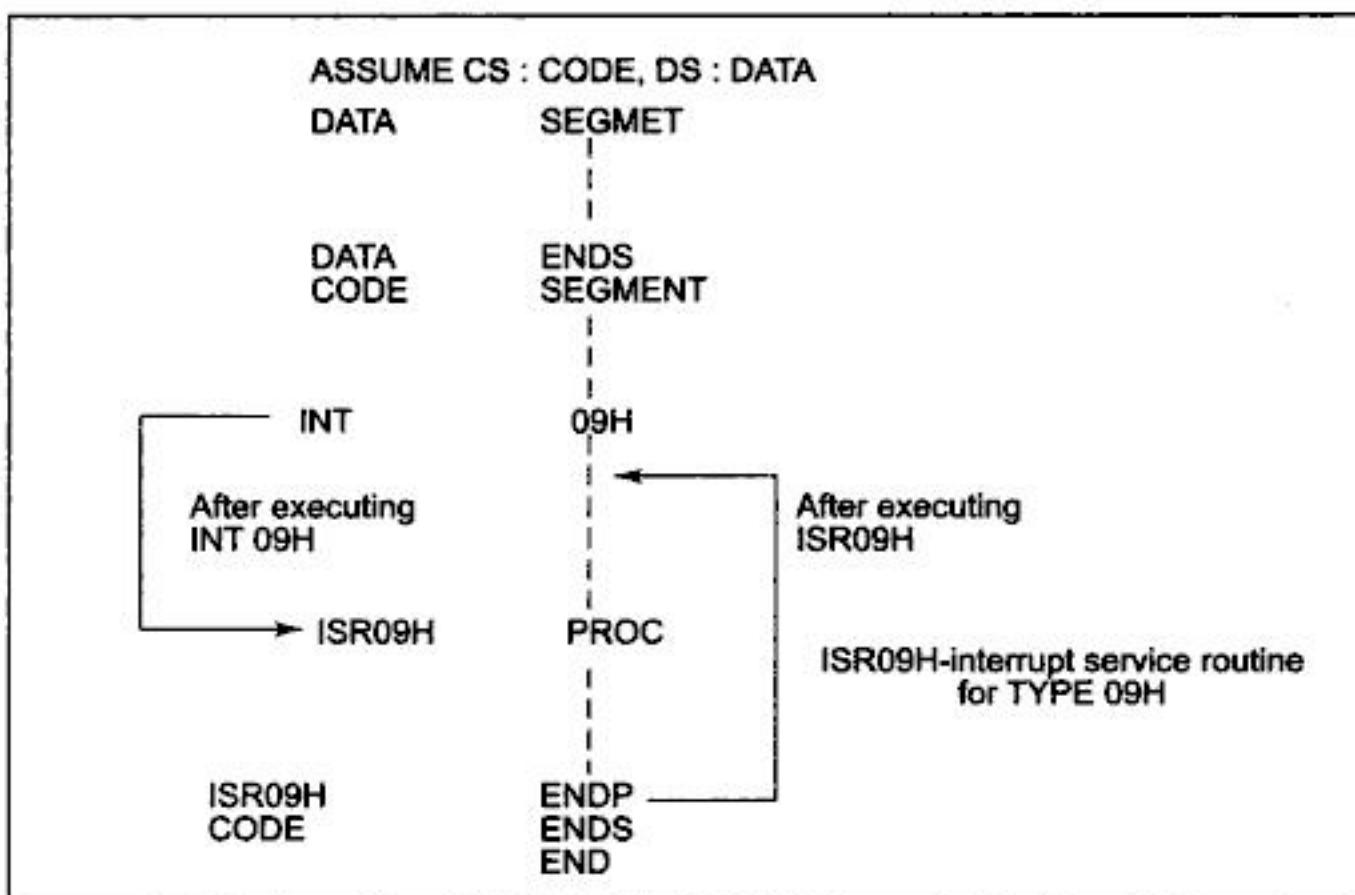
With the trailing edge of **LOCK**, the **INTA** goes low and remains low for two clock states before returning back to the high state.

It remains high till the start of the next machine cycle, i.e. next trailing edge of **ALE**.

Then **INTA** again goes low, remains low for two states before returning to the high state. The first trailing edge of **ALE** floats the bus **AD<sub>0</sub>-AD<sub>7</sub>**, while the second trailing edge prepares the bus to accept the type of the interrupt. The type of the interrupt remains on the bus for a period of two cycles.

## 4.7 INTERRUPT PROGRAMMING

While programming for any type of interrupt, the programmer must, either externally or through the program, set the interrupt vector table for that type preferably with the CS and IP addresses of the interrupt service routine. The method of defining the interrupt service routine for software as well as hardware interrupt is the same. Figure 4.7 shows the execution sequence in case of a software interrupt. It is assumed that the interrupt vector table is initialised suitably to point to the interrupt service routine. Figure 4.8 shows the transfer of control for the nested interrupts.



**Fig. 4.7 Transfer of Control during Execution of an Interrupt Service Routine**

### Program 4.3

Write a program to create a file **RESULT** and store in it 500H bytes from the memory block starting at 1000:1000, if either an interrupt appears at INTR pin with Type 0AH or an instruction equivalent to the above interrupt is executed.

**Note:** Pin **IRQ<sub>2</sub>** available at IO channel of PC is equivalent to Type 0AH interrupt.

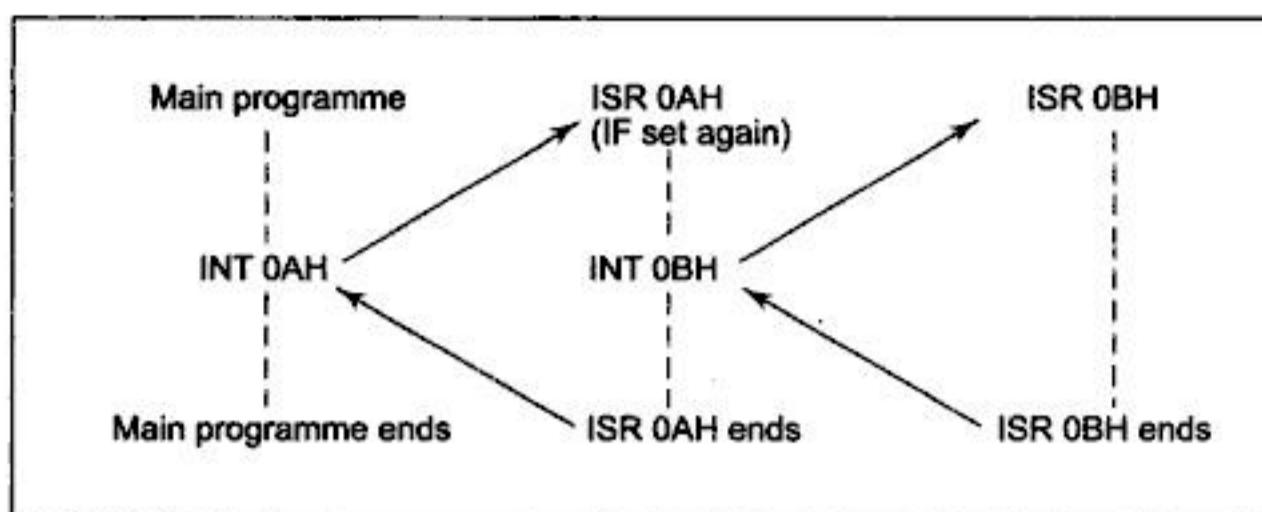


Fig. 4.8 Transfer of Control for Nested Interrupts

```

ASSUME CS : CODE, DS : DATA
DATA SEGMENT
  FILENAME DB "RESULT", "$"
  MESSAGE DB "FILE WASN'T CREATED SUCCESSFULLY", OAH, ODH, "$"
DATA ENDS
CODE SEGMENT
START: MOV AX, CODE
        MOV DS, AX          ; Set DS at CODE for setting IVT.
        MOV DX, OFFSET ISROA ; Set DX at the offset of ISROA.
        MOV AX, 250AH         ; Set IVT using function value 250AH in AX
        INT 21H              ; under INT21H.
        MOV DX, OFFSET FILENAME ; Set pointer to Filename.
        MOV AX, DATA          ; Set the DS at DATA for Filename
        MOV DS, AX
        MOV CX, 00H
        MOV AH, 3CH           ; Create file with the File name 'RESULT'.
        INT 21H
        JNC FURTHER          ; If no carry, create operation is
        MOV DX,OFFSET MESSAGE ; successful else
        MOV AH,09H             ; display the MESSAGE.
        INT 21H
        JMP STOP
FURTHER : INT 0AH           ; If the file is created successfully,
STOP :   MOV AH,4CH          ; write into it and return
        INT 21H              ; to DOS prompt.
                                ; This interrupt service routine writes 500 bytes into the
                                ; file RESULT and returns to the main program.

ISROA PROC NEAR
        MOV BX, AX            ; Take file handle in BX,
        MOV CX, 500H           ; byte count in CX,
        MOV DX, 1000H          ; offset of block in DX,
        MOV AX, 1000H          ; Segment value of block
        MOV DS, AX             ; in DS.
  
```

```

MOV AH, 40 H      ; Write in the file and
INT 21 H          ; return.
IRET              ;
ISROA    ENDP
CODE     ENDS
END START

```

#### Program 4.3 Listing

To execute the above program, first assemble it using MASM.EXE, link it using LINK.EXE. Then execute the above program at a DOS prompt. After execution, you will find a new file RESULT in the directory. Then apply an external pulse to IRQ2 pin of the IBM PC IO channel. This will again cause the execution of ISR that writes 500 H bytes into the file. For further details of the DOS function calls under INT 21H, refer the MSDOS Encyclopedia or MS-DOS Technical Reference.

#### Program 4.4

Write a program that gives display 'IRT2 is OK' if a hardware signal appears on  $\text{IRQ}_2$  pin and 'IRT3 is OK' if it appears on  $\text{IRQ}_3$  pin of PC IO Channel.

```

ASSUME CS:CODE, DS:DATA
DATA SEGMENT
MSG1 DB "IRT2 is OK",0AH,0DH,"$"
MSG2 DB "IRT3 IS OK",0AH,0DH,"$"
DATA ENDS
CODE ES

START: MOV AX, CODE
       MOV DS, AX           ; Set IVT for Type 0AH
       MOV DX, OFFSET ISR1
       MOV AX,250AH          ;  $\text{IRQ}_2$  is equivalent to Type 0AH
       INT 21H
       MOV DX, OFFSET ISR2          ; Set IVT for Type 0BH
       MOV AX, 250BH          ;  $\text{IRQ}_3$  is equivalent to TYPE 0BH
       INT 21H
HERE : JUMP HERE          ; ISR1 and ISR2 dispaly the message

ISR1  PROC LOCAL
       MOV AX, DATA
       MOV DS, AX
       MOV DX, OFFSET MSG1      ; Display message MSG1
       MOV AH, 09H
       INT 21 H
       IRET
ISR1  ENDP
ISR2  PROC LOCAL
       MOV AX, DATA
       MOV DS, AX

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

COUNT EQU 10H
DATA ENDS
CODE SEGMENT
    START :    MOV AX,DATA
                MOV DS,AX
                *
                *
                CALL ROUTINE
                *
                *
                *
PROCEDURE ROUTINE NEAR
    MOV BX,NUM
    MOV CX,COUTN
    *
ROUTINE      ENDP
CODE         ENDS
END START

```

Stack memory can also be used to pass parameters to a procedure. A main program may store the parameters to be passed to a procedure in its CPU registers. The registers will further be pushed on to the stack. The procedure during its execution pops back the appropriate parameters as and when required. This procedure of popping back the parameters must be implemented carefully because besides the parameters to be passed to the procedure the stack contains other important information like contents of other pushed registers, return addresses from the current procedure and other procedure or interrupt service routines.

#### Example 4.4

```

ASSUME CS:CODE, SS:STACK
CODE SEGMENT
    START :    MOV AX,STACK
                MOV SS,AX
                MOV AX,5577H
                MOV BX,2929H
                *
                PUSH AX
                PUSH BX
                CALL ROUTINE ; Decrements SP by 2 (by 4 far routine)
                *
                *
PROCEDURE ROUTINE NEAR
                *
                *
                MOV DX,SP
                ADD SP,02    ; Leave initial two stack bytes of return offset and
                            segment address after executing subroutine

```

```
POP BX      ; The data is
POP AX      ; Passes in BX,AX
MOV SP,DX
*
*
*
STACK SEGMENT
STACKDATA DB 200H DUP (?)
STACK ENDS
```

---

For passing the parameters to procedures using the PUBLIC & EXTRN directives, must be declared PUBLIC (for all routines) in the main routine and the same should be declared EXTRN in the procedure. Thus the main program can pass the PUBLIC parameter to a procedure in which it is declared EXTRN (external)

---

#### Example 4.5

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
PUBLIC NUMBER EQU 200H
    DATA ENDS
    CODE SEGMENT
START : MOV AX,DATA
        MOV DS,AX
        *
        *
        *
CALL ROUTINE
        *
        *
        *
PROCEDURE ROUTINE NEAR
EXTRN NUMBER
    MOV AX,NUMBER
    *
    *
    *
ROUTINE    ENDP
```

---

#### 4.9 HANDLING PROGRAMS OF SIZE MORE THAN 64K

As already discussed in Chapter 1, the maximum size of an 8086 segment is 64 KB. The same limitation is applicable to a code segment that contains executable program code. This obviously puts limitation on the maximum size of a program and thus how to write programs of size more than 64 K is going to be an interesting question, which is addressed in this section.

Unfortunately there is no technique to estimate the size of an executable program before it is assembled and linked. Thus one cannot come to know the physical byte size of a memory segment program while it is being developed. However, premeditating the assembly language program for a



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

START :      MOV AX,DATA
              MOV DS,AX
              .
              .
              CALL FAR_PTR ROUTINE1
              .
              .
              CALL FAR_PTR ROUTINE2
              .
              .

CODE1        ENDS
PROCEDURE     ROUTINE1 FAR
              .
              .
              .
ROUTINE1      ENDP
PROCEDURE     ROUTINE2 FAR
              .
              .
ROUTINE2      ENDP
END          START

```

#### 4.10 MACROS

Till now, we have studied the stack, subroutines, interrupts and interrupt service routines. It is a notable point that the control is transferred to a subroutine or an interrupt service routine whenever it is called or an interrupt signal appears at the interrupt pin of the processor. After executing these routines the control is again transferred back to the main calling program. Hence rather than writing a complete routine again and again, one may call it as many times as required. This imparts flexibility in programming as well as ease of troubleshooting. The concept of subroutine as well as interrupt service routine can be compared with an office where the main (calling) program acts as a head while the subroutines and interrupt service routines act as subordinates. The head may ask his subordinates to work out a particular task and be ready with the results. Here the main program calls subroutines and interrupt service routines and may refer the results of their execution for further processing. The subroutines and interrupt service routines are assigned labels for references.

The macro is also a similar concept. Suppose, a number of instructions are repeating through in the main program, the listings becomes lengthy. So a macro definition, i.e. a label, is assigned with the repeatedly appearing string of instructions. The process of assigning a label or macroname to the string is called defining a macro. A macro within a macro is called a nested macro. The macroname or macro definition is then used throughout the main program to refer to that string of instructions.

The difference between a macro and a subroutine is that in the macro the complete code of the instructions string is inserted at each place where the macro-name appears. Hence the EXE file becomes lengthy. Macro does not utilise the service of stack. There is no question of transfer of control as the program using the macro inserts the complete code of the macro at every reference of the macroname. On the other hand, subroutine is called whenever necessary, i.e. the control of execution is transferred to the subroutine, every time it is called. The executable code in case of the subroutines becomes

smaller as the subroutine appears only once in the complete code. Thus, the EXE file is smaller as compared to the program using macro. The control is transferred to a subroutine whenever it is called, and this utilizes the stack service. The program using subroutine requires less memory space for execution than that using macro. Macro requires less time for execution, as it does not contain CALL and RET instructions as the subroutines do.

#### 4.10.1 Defining a MACRO

A MACRO can be defined anywhere in a program using the directives MACRO and ENDM. The label prior to MACRO is the macro name which should be used in the actual program. The ENDM directive marks the end of the instructions or statements sequence assigned with the macro name. The following macro DISPLAY displays the message MSG on the CRT. The syntax is as given:

```
DISPLAY MACRO
    MOV AX, SEG MSG
    MOV DS, AX
    MOV DX, OFFSET MSG
    MOV AH, 09 H
    INT 21 H
ENDM
```

The above definition of a macro assigns the name DISPLAY to the instruction sequence between the directives MACRO and ENDM. While assembling, the above sequence of instructions will replace the label 'DISPLAY', whenever it appears in the program.

A macro may also be used in a data segment. In other words, a macro may also be used to represent statements and directives. The concept of macro remains the same independent of its contents. The following example shows a macro containing statements. The macro defines the strings to be displayed.

```
STRINGS MACRO
    MSG1 DB 0AH,0DH, "Program terminated normally",0AH,0DH, "$"
    MSG2 DB 0AH,0DH, "Retry , Abort, Fail",0AH,0DH, "$"
ENDM
```

A macro may be called by quoting its name, along with any values to be passed to the macro. Calling a macro means inserting the statements and instructions represented by the macro directly at the place of the macroname in the program.

#### 4.10.2 Passing Parameters to a MACRO

Using parameters in a definition, the programmer specifies the parameters of the macro those are likely to be changed each time the macro is called. For example, the DISPLAY macro written in Section 4.10.1 can be made to display two different messages MSG1 and MSG2, as shown.

```
DISPLAY MACRO MSG
    MOV AX, SEG MSG
    MOV DS, AX
    MOV DX, OFFSET MSG
    MOV AH, 09 H
    INT 21 H
ENDM
```

This parameter MSG can be replaced by MSG1 or MSG2 while calling the macro as shown.

```
        DISPLAY MSG1  
        .  
        DISPLAY MSG2  
        .  
        .  
MSG1 DB OAH,ODH, "Program Terminated Normally", OAH,ODH, "$"  
MSG2 DB OAH,ODH, "Retry, Abort, Fail", OAH,ODH, "$"
```

There may be more than one parameter appearing in the macro definition, meaning thereby that there may be more than one parameters to be passed to the macro, and each of them is liable to be changed. All the parameters are specified in the definition sequentially and also in the call with the same sequence.

A macro may be defined in another macro or in other words a macro may be called from inside a macro. This type of macro is called a nested macro. All the directives available in MASM can also be used in a macro and carry the same significance.

#### 4.11 TIMINGS AND DELAYS

It is obvious from the studies of the timing diagrams that every instruction requires a definite number of clock cycles for its execution. Thus every instruction requires a fixed amount of time, i.e. multiplication of the number of clock cycles required for the execution of the instruction and the period of the clock at which the microprocessor is running. The duration required for the execution of an instruction can be used to derive the required delays. A sequence of instructions, if executed by a microprocessor, will require a time duration that is the sum of all the individual time durations required for execution of each instruction. Note that in a loop program, the number of instructions in the program may be less but the number of instructions actually executed by the microprocessor depend on the loop count. Also in case of subroutines and interrupt service routines the actual number of instructions executed by the microprocessor depends on the procedure or interrupt service routine length along with the main calling program. The required number of clock states for execution of each instruction of 8086/88 are given in Appendix A.

The procedure of generating delays using a microprocessor based system can be stepwise described as follows.

1. Determine the exact required delay.
2. Select the instructions for delay loop. While selecting the instructions, care should be taken that the execution of these instructions does not interfere with the main program execution. In other words, any memory location or register used by the main program must not be modified by the delay routine. The instructions executed for the delay loop are dummy instructions in the sense that the result of those instructions is useless but the time required for their execution is an elemental part of the required delay.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The procedure will be clearer while solving problems on memory interfacing with 8086/88.

As a good and efficient interfacing practice, the address map of the system should be continuous as far as possible, i.e. there should be no windows in the map and no fold back space should be allowed. A memory location should have a single address corresponding to it, i.e. absolute decoding should be preferred, and minimum hardware should be used for decoding. In a number of cases, linear decoding may be used to minimise the required hardware.

Let us now consider a few example problems on memory interfacing with 8086.

### Problem 5.1

Interface two  $4K \times 8$  EPROMS and two  $4K \times 8$  RAM chips with 8086. Select suitable maps.

**Solution** We know that, after reset, the IP and CS are initialised to form address FFFF0H. Hence, this address must lie in the EPROM. The address of RAM may be selected anywhere in the 1MB address space of 8086, but we will select the RAM address such that the address map of the system is continuous, as shown in Table 5.1.

**Table 5.1 Memory Map for Problem 5.1**

Address	$A_{19}$	$A_{18}$	$A_{17}$	$A_{16}$	$A_{15}$	$A_{14}$	$A_{13}$	$A_{12}$	$A_{11}$	$A_{10}$	$A_{09}$	$A_{08}$	$A_{07}$	$A_{06}$	$A_{05}$	$A_{04}$	$A_{03}$	$A_{02}$	$A_{01}$	$A_{00}$
FFFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
EPROM																		$8K \times 8$		
FE000H	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
FDFFFFH	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
RAM																		$8K \times 8$		
FC000H	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

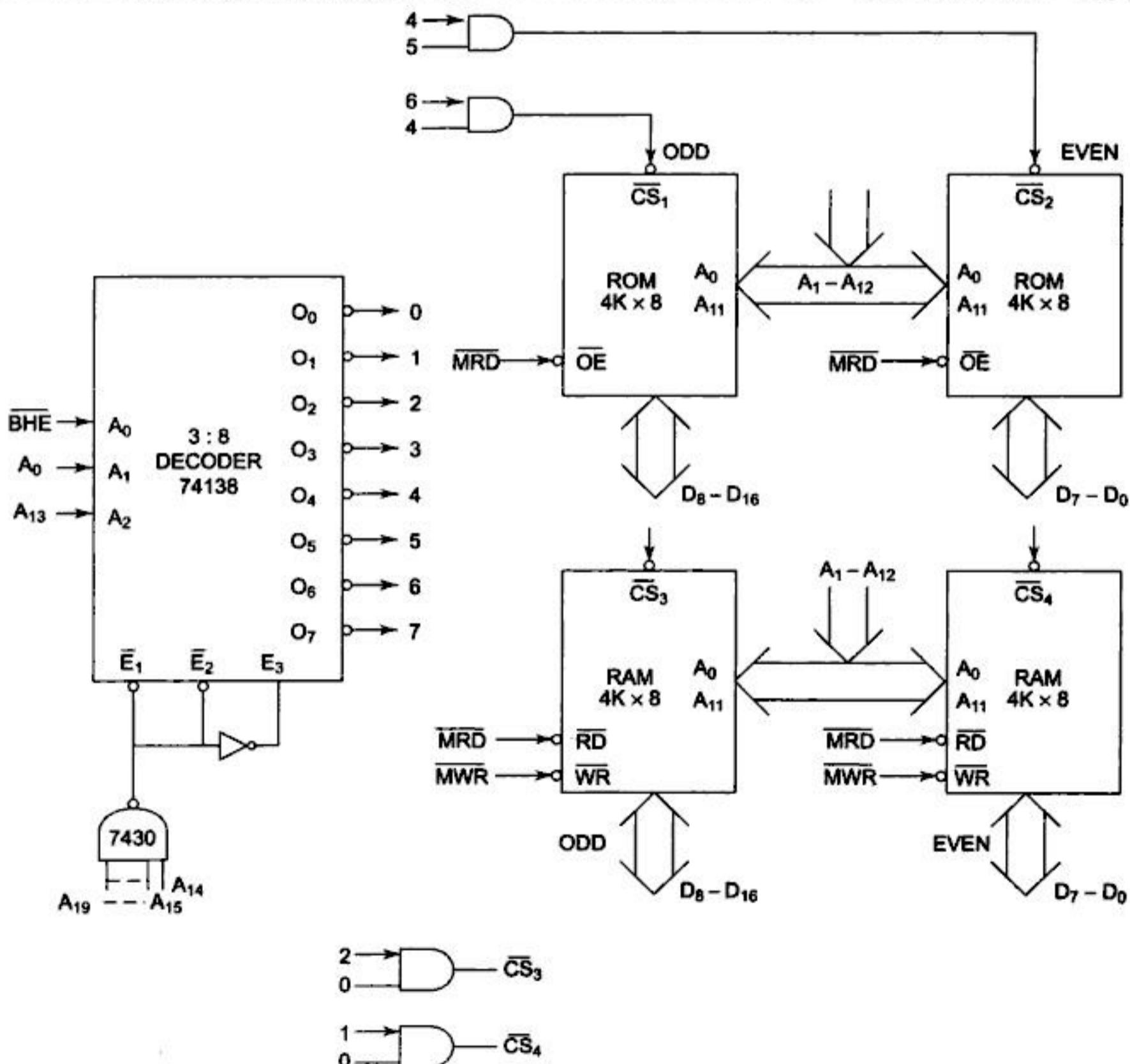
Total 8K bytes of EPROM need 13 address lines  $A_0 - A_{12}$  (since  $2^{13} = 8K$ ). Address lines  $A_{13} - A_{19}$  are used for decoding to generate the chip select. The  $\overline{BHE}$  signal goes low when a transfer is at odd address or higher byte of data is to be accessed. Let us assume that the latched address,  $\overline{BHE}$  and demultiplexed data lines are readily available for interfacing. Figure 5.1 shows the interfacing diagram for the memory system.

The memory system in this example contains in total four  $4K \times 8$  memory chips.

The two  $4K \times 8$  chips of RAM and ROM are arranged in parallel to obtain 16-bit data bus width. If  $A_0$  is 0, i.e. the address is even and is in RAM, then the lower RAM chip is selected indicating 8-bit transfer at an even address. If  $A_0$  is 1, i.e. the address is odd and is in RAM, the  $\overline{BHE}$  goes low, the upper RAM chip is selected, further indicating that the 8-bit transfer is at an odd address. If the selected addresses are in ROM, the respective ROM chips are selected. If at a time  $A_0$  and  $\overline{BHE}$  both are 0, both the RAM or ROM chips are selected, i.e. the data transfer is of 16 bits. The selection of chips here takes place as shown in Table 5.2.

**Table 5.2 Memory Chip Selection for Problem 5.1**

<i>Decoder I/P →</i> <i>Address/BHE →</i>	<i>A<sub>2</sub></i>	<i>A<sub>1</sub></i>	<i>A<sub>0</sub></i>	<i>BHE</i>	<i>Selection/Comment</i>
Word transfer on $D_0 - D_{15}$	0	0	0		Even and odd addresses in RAM
Byte transfer on $D_7 - D_0$	0	0	1		Only even address in RAM
Byte transfer on $D_8 - D_{15}$	0	1	0		Only odd address in RAM
Word transfer on $D_0 - D_{15}$	1	0	0		Even and odd addresses in ROM
Byte transfer on $D_0 - D_7$	1	0	1		Only even address in ROM
Byte transfer on $D_8 - D_{15}$	1	1	0		Only odd address in ROM

**Fig. 5.1 Interfacing Problem 5.1**

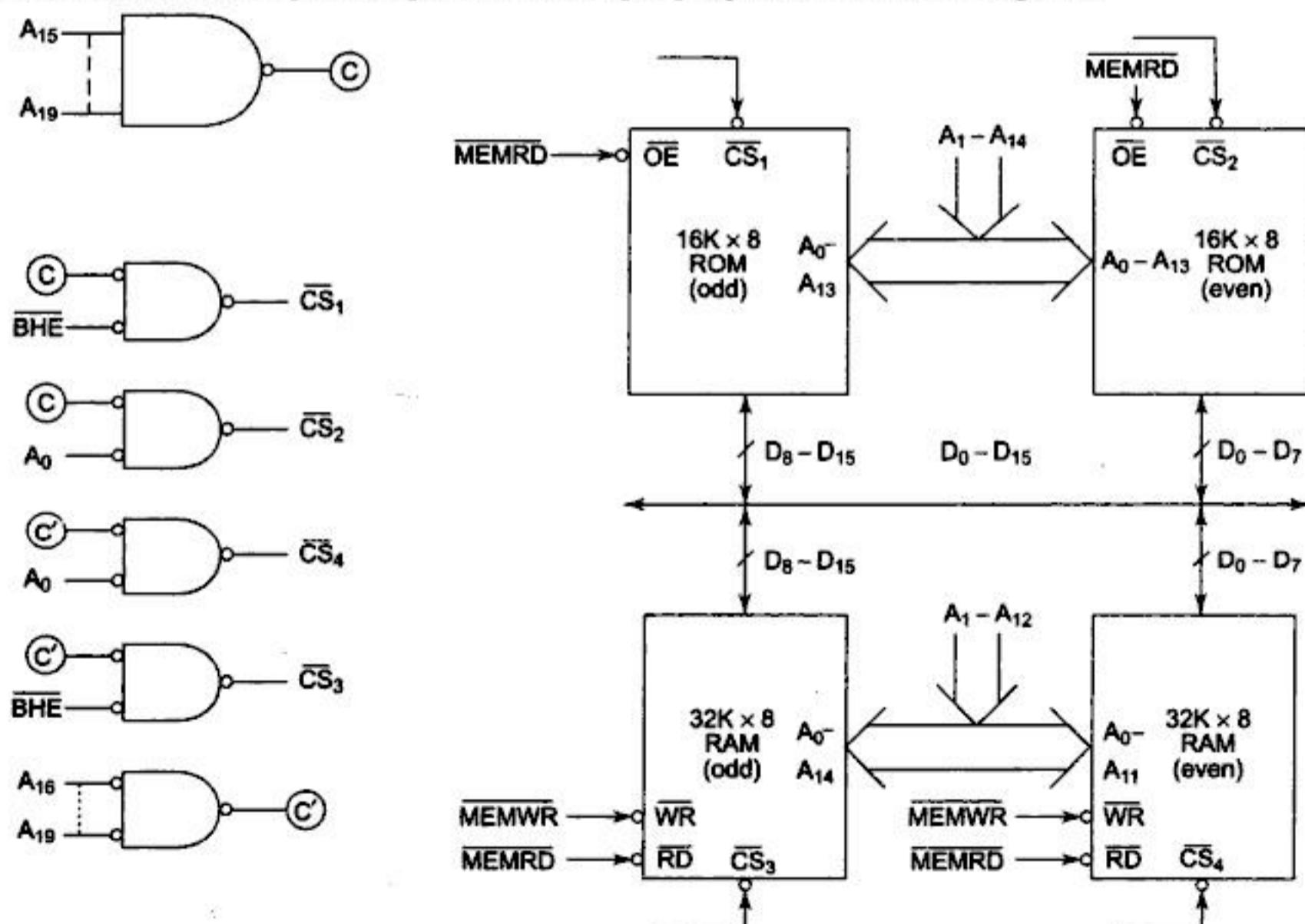
### Problem 5.2

Design an interface between 8086 CPU and two chips of  $16K \times 8$  EPROM and two chips of  $32K \times 8$  RAM. Select the starting address of EPROM suitably. The RAM address must start at  $00000H$ .

**Solution:** The last address in the map of 8086 is FFFFFH. After resetting, the processor starts from FFFF0H. Hence this address must lie in the address range of EPROM. Figure 5.2 shows the interfacing diagram, and Table 5.3 shows complete map of the system.

**Table 5.3** Address Map for Problem 5.2

It is better not to use a decoder to implement the above map because it is not continuous, i.e. there is some unused address space between the last RAM address (0FFFFH) and the first EPROM address (F8000H). Hence the logic is implemented using logic gates, as shown in Fig. 5.2.



**Fig. 5.2** Interfacing Problem 5.2



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The implementation of the above map is shown in Fig. 5.3 using the same technique as in Problem 5.1 and Problem 5.2. All the address, data and control signals are assumed to be readily available.

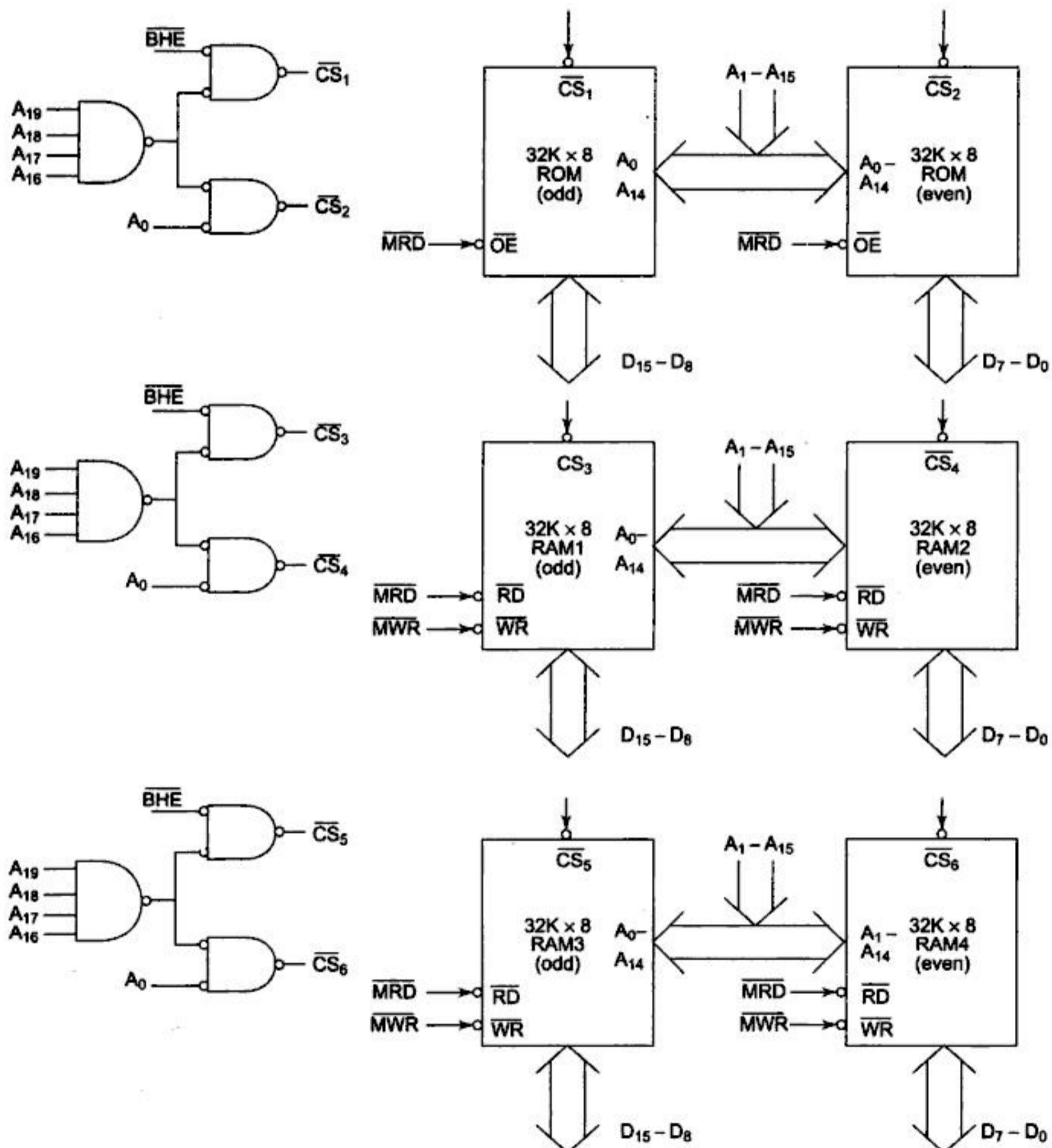


Fig. 5.3 Interfacing Problem 5.3



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

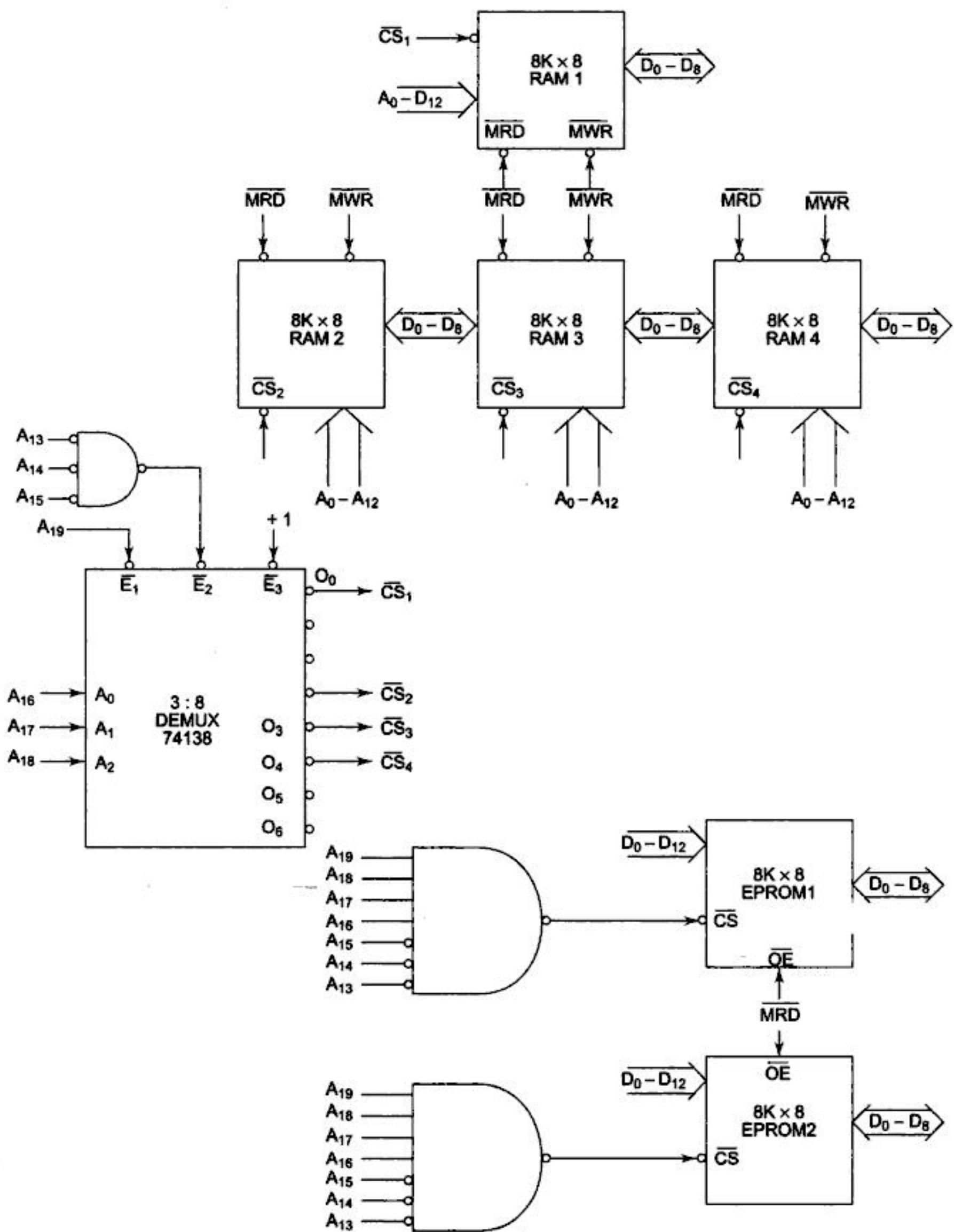


Fig. 5.4 Interfacing Problem 5.4



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

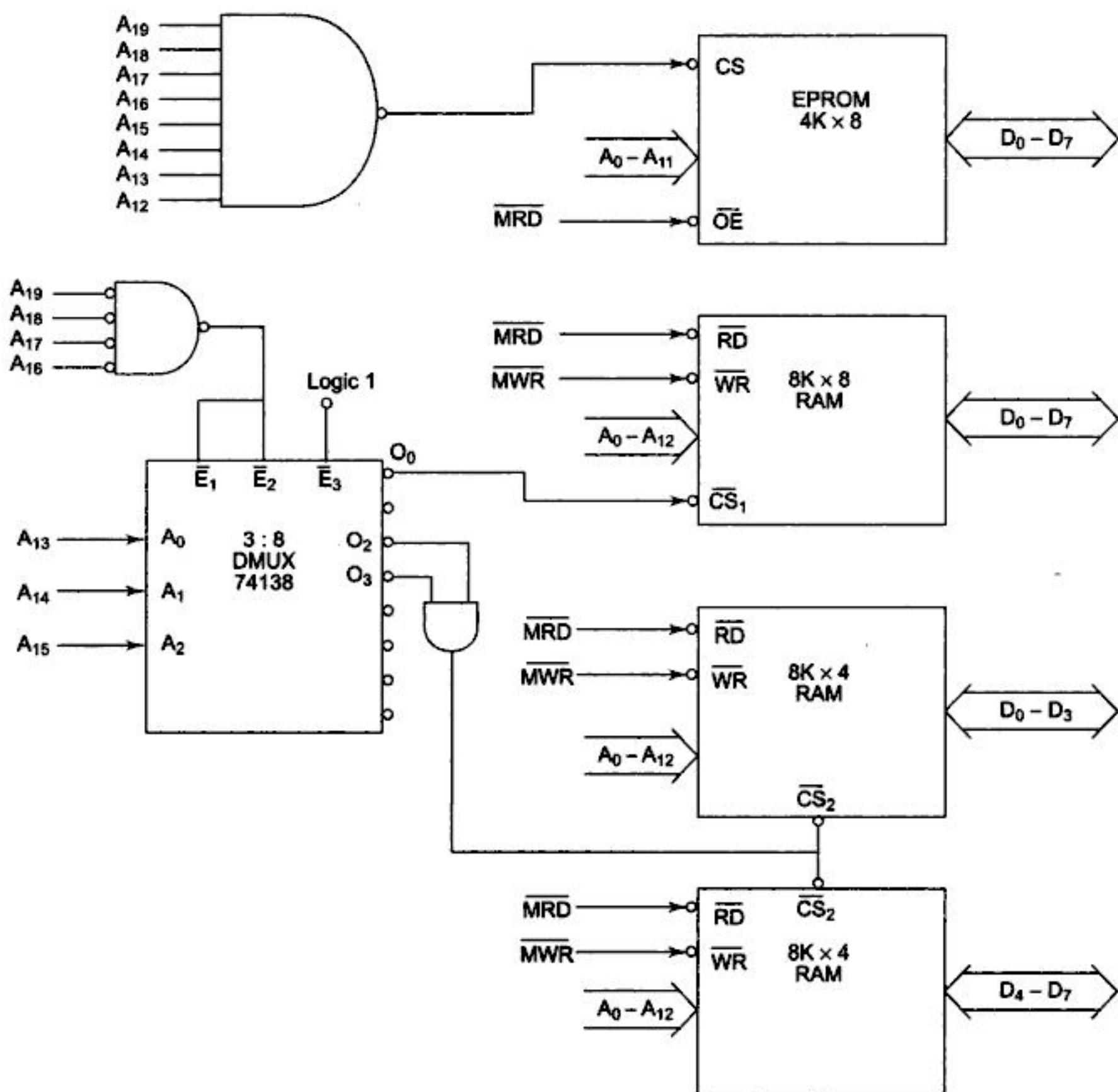


Fig. 5.6 Alternative Implementation for Problem 5.5

This activity is similar to reading the data from each cell of the memory, independent of the requirement of microprocessor, regularly. During this refresh period all other operations (accesses) related to the memory subsystem are suspended. Hence the refresh activity causes loss of time, resulting in reduced system performance. However, keeping in view the advantages of dynamic RAM, like low power consumption, higher packaging density and low cost, most of the advanced computer systems are designed using dynamic RAMs, of course, at the cost of operating speed. Also, the refresh mechanism and the additional hardware required makes the interfacing hardware, in case of dynamic RAM, more complicated, as compared to static RAM interfacing circuit. A dedicated hardware chip called as dynamic RAM controller is the most important part of the interfacing circuit.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

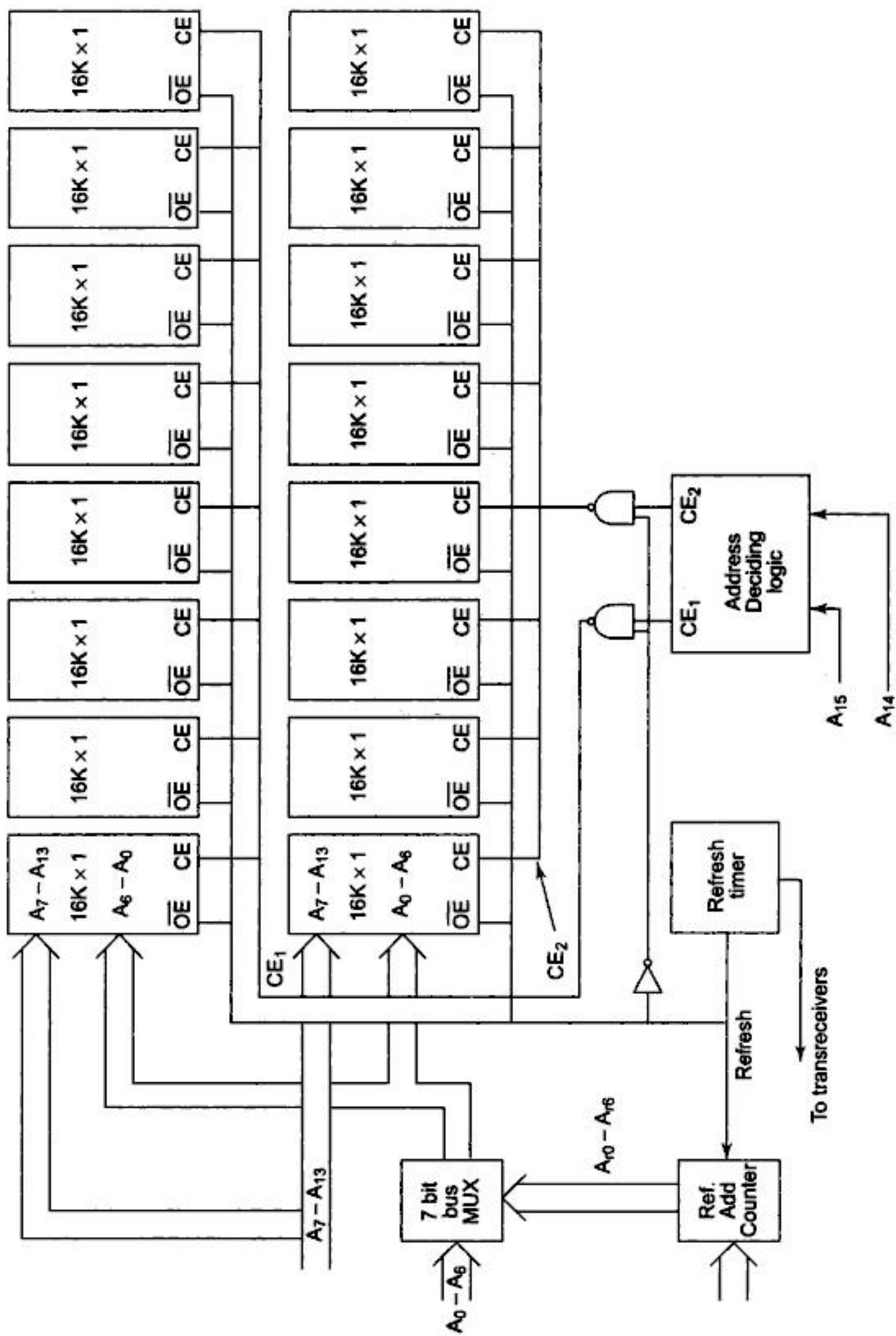


Fig. 5.7 Dynamic RAM Refreshing Logic

while high  $\overline{OE}$  prevents the data from appearing on the data bus, as said in the description of the memory refresh cycle. The  $16K \times 1$ -bit dynamic RAM has an internal array of  $128 \times 128$  cells, requiring 7 bits for row addresses. The lower order seven lines  $A_0 - A_6$  are multiplexed with the refresh counter output  $A_{10} - A_{16}$ . The logic is shown in Fig. 5.7.

The pin assignments for 2164 dynamic RAM is shown in Fig. 5.8 (a). The  $\overline{RAS}$  and  $\overline{CAS}$  are row and column address strobes and are driven by the dynamic RAM controller outputs.  $A_0 - A_7$  lines are the row or column address lines, driven by  $OUT_0 - OUT_7$ , outputs of the controller. The  $\overline{WE}$  pin indicates memory write cycles. The  $D_{IN}$  and  $D_{OUT}$  pins are data pins for write and read operations, respectively. In practical circuits, the refreshing logic is integrated inside dynamic RAM controller chips like 8203, 8202 and 8207, etc.

Intel's 8203 is a dynamic RAM controller (Fig. 5.8(b)) that supports 16K or 64K dynamic RAM chips. This selection is done using pin 16K/64K. If it is high, the 8203 is configured to control 16K dynamic RAM, else it controls 64K dynamic RAM. The address inputs of 8203 controller accept address lines  $A_1$  to  $A_{16}$  on lines  $AL_0 - AL_7$  and  $AH_0 - AH_7$ . The  $A_0$  line is used to select the even or odd bank. The  $\overline{RD}$  and  $\overline{WR}$  signals decode whether the cycle is a memory read or memory write cycle and are accepted as inputs to 8203 from the microprocessor. The  $\overline{WE}$  signal specifies the memory write cycle and is an output from 8203 that drives the  $\overline{WE}$  input of dynamic RAM memory chip. The  $OUT_0 - OUT_7$ , set of eight pins is an 8-bit output bus that carries multiplexed row and column addresses of a bit location in a dynamic RAM chip. The  $\overline{CAS}$  signal strobes the column address on  $OUT_0 - OUT_7$ . The  $RAS_1 - RAS_0$  pins strobes row address on  $OUT_0 - OUT_7$ , for at the most two banks of 2164 dynamic RAM chips. Actually the row and column addresses are derived from the address lines  $A_1 - A_{16}$  accepted by the controller on its inputs  $AL_0 - AL_7$  and  $AH_0 - AH_7$ . An external crystal may be applied between  $X_0$  and  $X_1$  pins, otherwise, with the  $OP_2$  pin at +12V, a clock signal may be applied at the pin CLK. The  $\overline{PCS}$  pin accepts the chip select signal derived by an address decoder.

The  $\overline{REFREQ}$  pin is used whenever the memory refresh cycle is to be initiated by an external signal. The  $\overline{XACK}$  signal indicates that data is available during a read cycle or it has been written if it is a write cycle. It can be used as a strobe for data latches or as a ready signal to the processor. The  $\overline{SACK}$  output signal marks the beginning of a memory access cycle. If a memory request is made during a memory refresh cycle, the  $\overline{SACK}$  signal is delayed till the starting of memory read or write cycle. Figure 5.9 shows how the 8203 can be used to control a 256K bytes memory subsystem for a maximum mode 8086 microprocessor system. This design assumes that data and address busses are inverted and latched, hence the inverting buffers and inverting latches are used (8283-inverting buffer and 8287-inverting latch).

Figure 5.10 shows the interfacing of 512K byte dynamic RAM with 8086 using an advanced dynamic RAM controller 8208. Most of the functions of 8208 and 8203 are similar but 8208 can be used to refresh the dynamic RAM using DMA approach. The memory system is divided into even and odd banks of 256Kbyte each, as required for an 8086 system. The inverted  $\overline{AACK}$  output of 8208 latches the  $A_0$  and  $\overline{BHE}$  signals required for selecting the banks. If the latched bank select signal and the  $\overline{WE}$ /PCLK output of 8208 both go low it indicates a write operation to the respective bank.

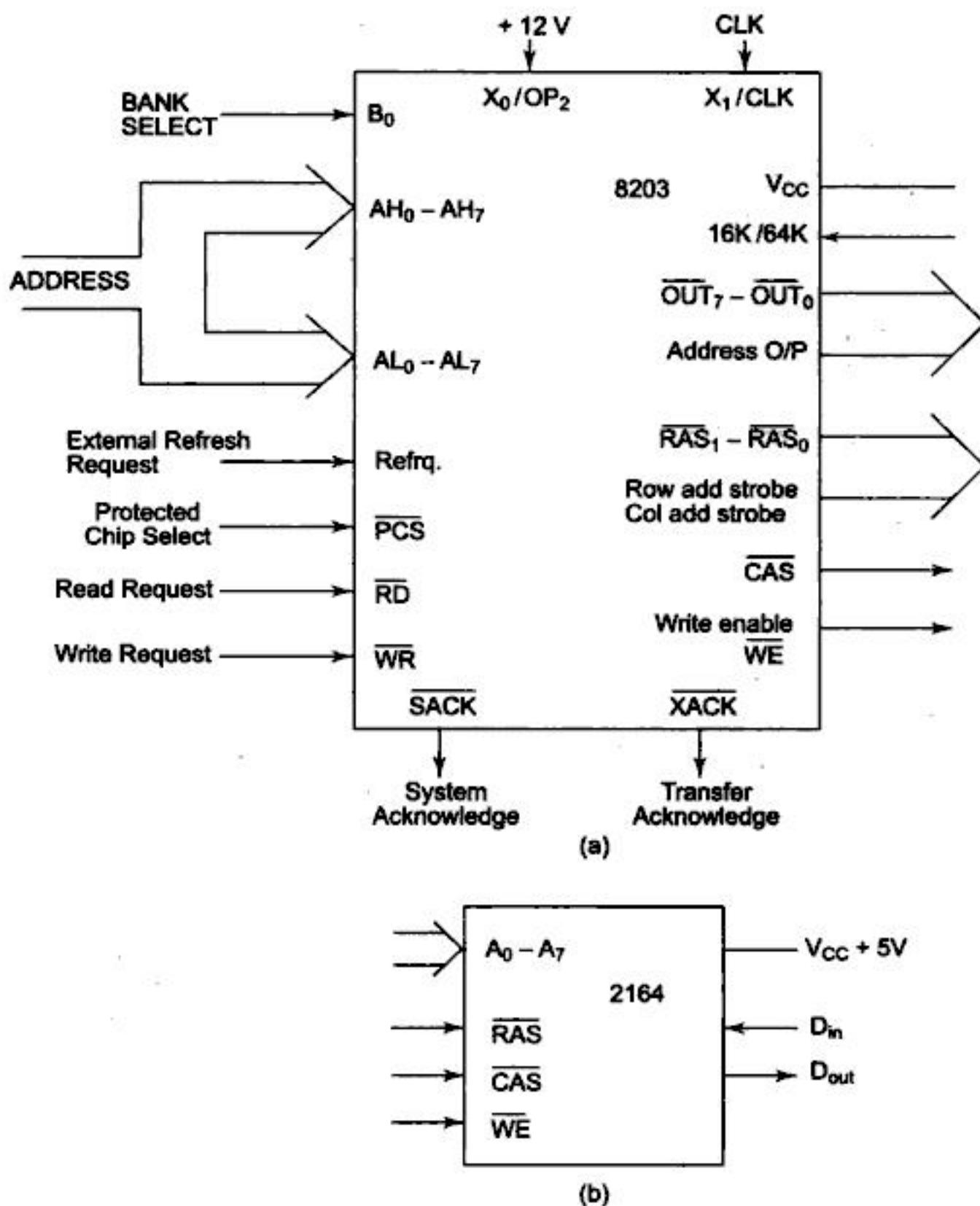


Fig. 5.8 (a) Dynamic RAM Controller (b) 1-bit Dynamic RAM

### 5.3 INTERFACING I/O PORTS

I/O ports or Input/Output ports are the devices through which the microprocessor communicates with other devices or external data sources/destinations. Input activity, as one may expect, is the activity that enables the microprocessor to read data from external devices, for example keyboards, joysticks, mouse, etc. These devices are known as input devices as they feed data into a microprocessor system.

Output activity transfers data from the microprocessor to the external devices, for example CRT display, 7-segment displays, printers, etc. The devices which accept the data from a microprocessor system are called output devices. Thus for a microprocessor the input activity is similar to read operation,



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

while the output activity is similar to write operation. Note that an input device can only be read and an output device can only be written.

Hence  $\overline{\text{IORD}}$  operation is related with reading data from an input device and not an output device and  $\overline{\text{IOWR}}$  operation means writing data to an output device and not an input device. The control word and status word may be written and read respectively in both, input or output, devices, in case of programmable devices.

After executing an OUT operation, the data appears on the data bus and simultaneously a device select signal is generated from the address and control signals. Now, if the data is to be there, at the output of the device till the next change, it must be latched. Also if the output port is to source large currents the port lines must be buffered. Hence the latch acts as a good output port. The chip 74LS373, contains eight buffered latches and can be used as an 8-bit output port. While reading, an input device one must take care that much current should not be sourced or sunk from the data lines to avoid loading. To overcome this problem, one may use a tristate buffer as an input device. An input port may not be a latch as it reads the status of a signal at a particular instant. The chip 74LS245 contains eight buffers and may be used as an 8-bit input port. Actually, 74LS245 is a bidirectional buffer, but while using it as an input device, only one direction is useful. This direction of data transfer in 74LS245 is selected using its DIR pin. The pin diagrams of 74LS 373 and 74LS 245 are shown in Figs 5.11 (a) and (b). The  $\overline{\text{OE}}$  and  $\overline{\text{CS}}$  are the chip selects of 74LS373 and 74LS245 respectively. Ds and Qs are corresponding latch inputs and outputs respectively.

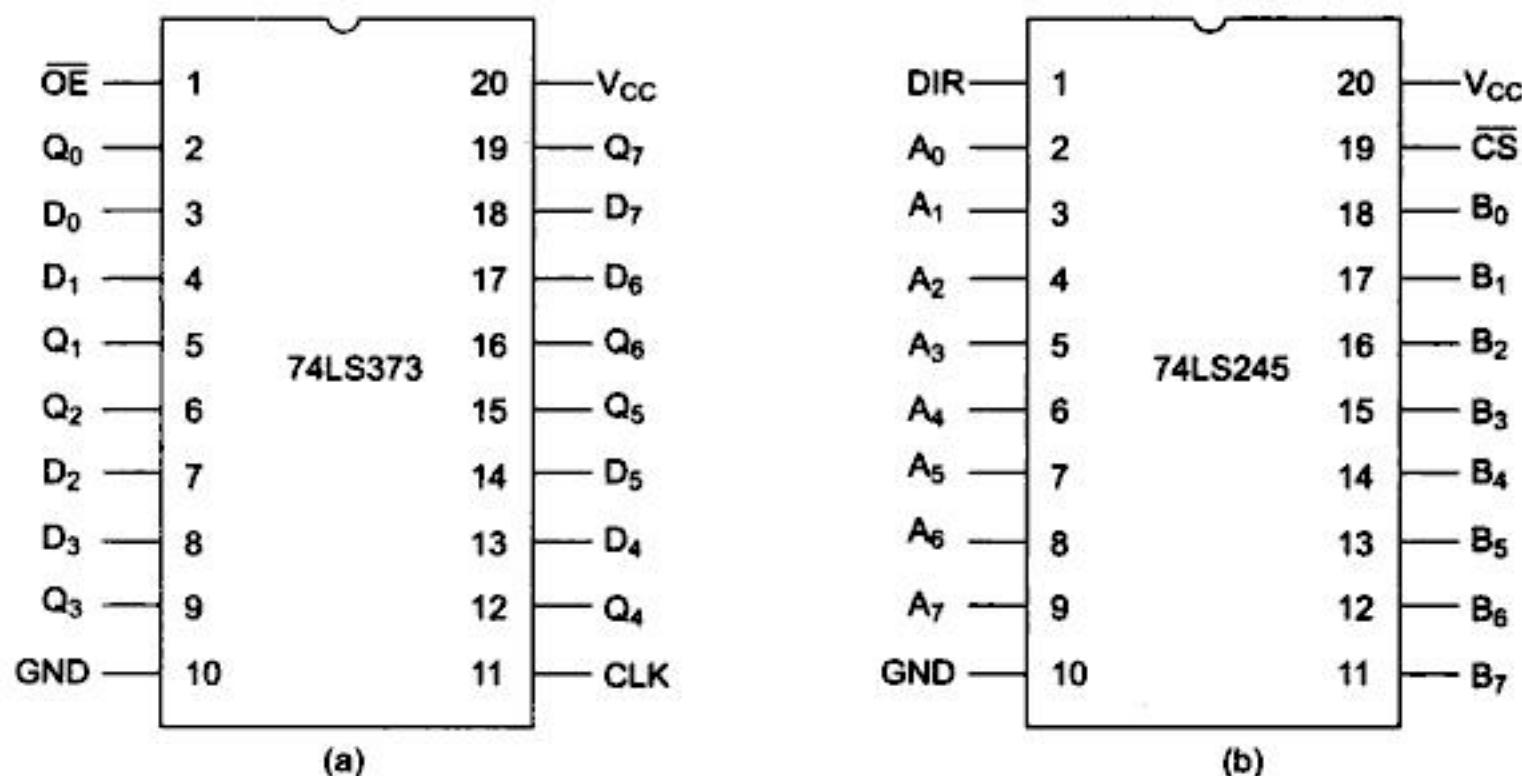


Fig. 5.11 (a) Latch (O/P Port) (b) Buffer (I/P Port)

The CLK pin is clock input for D flip-flops. If DIR is 1, then the direction is from A(I/Ps) to B(O/Ps), otherwise the data direction is from B(I/Ps) to A(O/Ps).

**Steps in Interfacing an I/O Device** The following steps are performed to interface a general I/O device with a CPU:

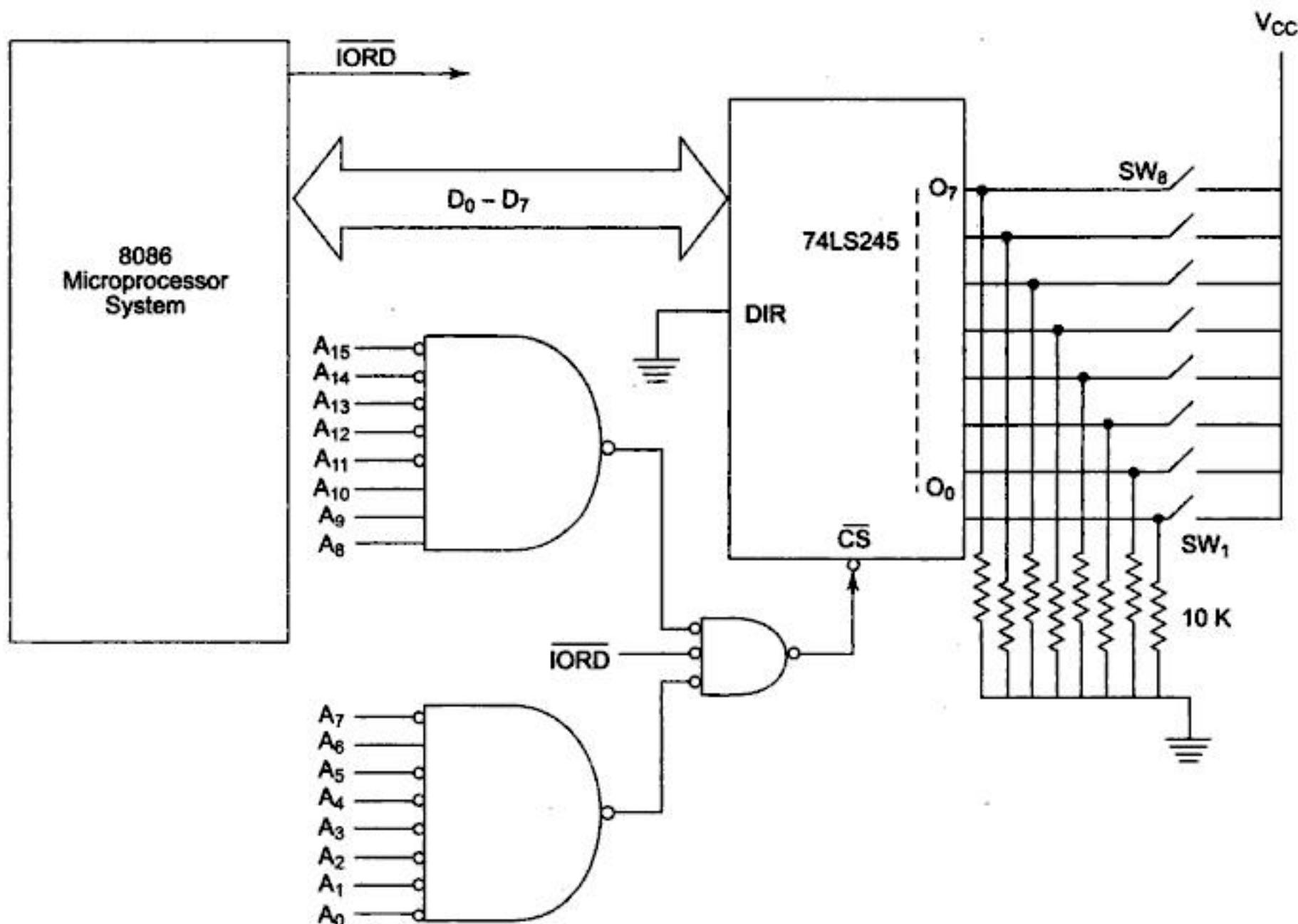


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Problem 5.6**

Interface an input port 74LS245 to read the status of switches SW<sub>1</sub> to SW<sub>8</sub>. The switches, when shorted, input a '1' else input a '0' to the microprocessor system. Store the status in register BL. The address of the port is 0740H.

**Solution** The hardware interface circuit is shown in Fig. 5.12. The address, control and data lines are assumed to be readily available at the microprocessor system.



**Fig. 5.12** Interfacing Input Port 74LS245

The ALP is given as follows:

```

MOV BL, 00H      ; Clear BL for status
MOV DX, 0740H    ; 16-bit port address in DX
IN AL, DX        ; Read port 0740H for switch positions
MOV BL, AL        ; Store status of switches from AL into BL
HLT              ; Stop

```

**Program 5.1**

ALP for Problem 5.6.

Here LSB bit of BL corresponds to the status of SW<sub>1</sub>, and likewise the MSB of BL corresponds to the

status of  $SW_8$ . The '1' indicates 'on' or shorted switch and the '0' indicates an 'off' or opened switch. The pull-up registers in Fig. 5.12 are necessary because the open switches should input a '0' to the system but the TTL port 74LS245 will read the free input as '1'. (Free TTL inputs are always read as logic '1').

### Problem 5.7

Design an interface of an input port 74LS245 to read the status of switches  $SW_1$  to  $SW_8$  (as in the previous problem), and an output port 74LS373 with 8086. Display the number of a key that is pressed, i.e. from 1 to 8 on a 7-seg display with help of the output port. Write an ALP for this task, assume that only one key is pressed at a time. Draw the schematic of the required hardware. The input port address is 0008H and the output port address is 000AH.

**Solution** In the previous problem, one might have noted that a lot of hardware is required to decode the port address absolutely. Thus instead of decoding the address completely, only a part of it may be decoded. For example, instead of using 16 address lines  $A_{15}-A_0$ , one may use only  $A_3-A_0$ . In this problem, the address 0008H may then be converted to  $xxx8H$ , where  $x$  denotes a don't care condition. Thus the port may have more than one address, for example 2358H, 1728H etc. Only the least significant nibble of the address needs to be 8H. The disadvantage of the scheme is that there are a number of addresses of the same port. Hence, the system must have only one port that has the lowest nibble address 8H, otherwise, the system may malfunction. Thus for smaller systems containing a few I/O ports, this scheme is suitable and advantageous as it requires less hardware.

The status of the switches is first read into the register AL. For displaying the shorted switch number in the 7-seg display, the bit corresponding to the switch is checked by rotating AL through carry and then checking the carry flag. If the carry flag is '1', after one left rotation, it means  $SW_1$  is on. If the carry flag sets after two rotations,  $SW_2$  is on and so on. Register CL is incremented after each rotation so that it contains the pressed switch number. The 8-bit contents of register CL are converted to 7-segment codes by a BCD to 7-seg decoder. The complete hardware (Fig. 5.13) and the ALP is given as shown. Note that both the ports are interfaced at even addresses, i.e. with lower order data bus  $D_0 - D_7$ .

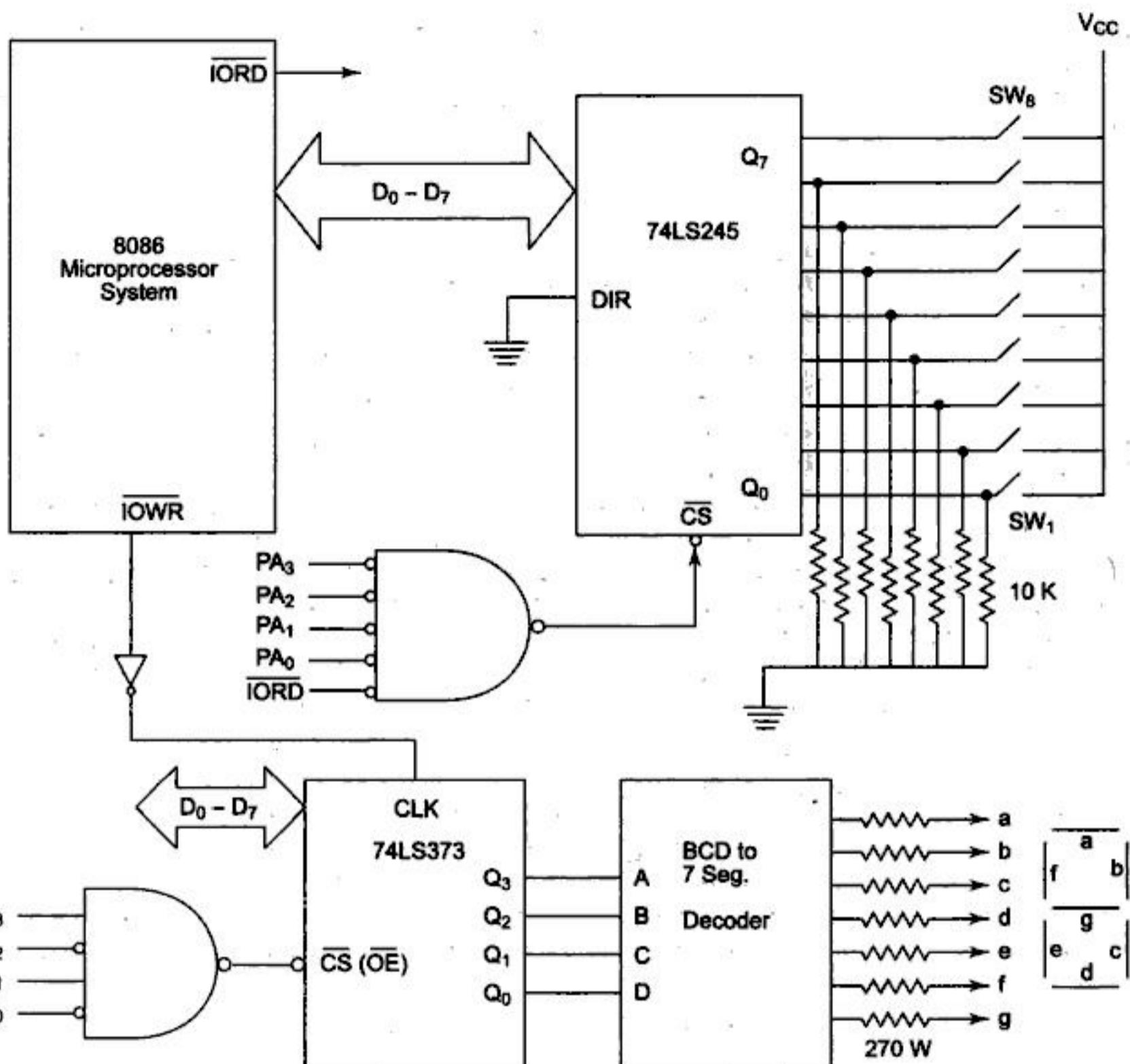
Common cathode displays are used along with corresponding BCD to 7-seg decoder.

```

MOV BL, 00 ; Clear BL for switch status
MOV CL, 00 ; Clear CL for switch number
XOR AX, AX ; Clear accumulators and flag
IN AL,08H ; Read switch status
INC CL ; Increment CL for 1st switch
YY: RCR AL ; Rotate switch status
      JC XX ; If carry, halt,
      INC CL ; else increment CL for next switch
      JMP YY ; number till carry is 1
XX: MOV AL,CL ; Take switch number into AL
OUT 0AH,AL ; Out BCD switch number for display
HLT ; Stop

```

Program 5.2 ALP for Problem 5.7



**Fig. 5.13** Interfacing Switches and Displays for Problem 5.7

### Problem 5.8

Using 74LS373 output ports and 7-segment displays, design a seconds counter that counts from 0 to 9. Draw the suitable hardware schematic and write an ALP for this problem. Assume that a delay of 1sec is available as a subroutine. Select the port address suitably.

**Solution** The counter hardware is shown in Fig. 5.14. Common cathode displays are used along with a suitable BCD to 7-segment decoder. The ALP calls the subroutine 'DELAY' that generates a delay program of 1sec. After counting from 0 to 9, it again starts from 0. The output port is interfaced at address 0008H.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Solution** Let us select the two port addresses 0004H and 0008H for the output ports. The first port 0004H outputs 7-seg code while the second output port 0008H selects the display by grounding the common cathode. The hardware is given in Fig. 5.15.

The 7-seg codes for C.C. displays can be decided as given. For a LED to be 'on', that particular anode should be 1 and the common cathode line should be grounded, using a port line that drives a transistor. Thus for the numbers to be displayed the code is calculated as shown.

Decimal no.	a $A_7$	b $A_6$	c $A_5$	d $A_4$	e $A_3$	f $A_2$	g $A_1$	dp $A_0$	
1-	1	1	0	0	0	0	0	0	= C0
2-	1	1	0	1	1	0	1	0	= DA
3-	1	1	1	1	0	0	1	0	= F2
4-	0	1	1	0	0	1	1	0	= 66
5-	1	0	1	1	0	1	1	0	= B6

These codes are stored in a look up table starting from 2000H:0000, as shown below.

2000 : 0000	→ C0 H
2000 : 0001	→ DA H
2000 : 0002	→ F2 H
2000 : 0003	→ 66 H
2000 : 0004	→ B6 H

Only one display should be selected at a time, i.e. only the corresponding bit of port 2 should be high for selecting a common cathode display. All the other bits should be low to keep the other displays disabled. Thus to enable the least significant display, the LSB of the 8-bit selected port should remain '1'. Hence AL should have 01 or E1H in it to select the least significant display. The codes for the selection of displays and 7-segment codes directly depend upon the hardwired connections between them.

```

MOV AX, 2000H      ; Initialize pointer to
MOV DS, AX          ; Code table DS:BX
MOV BX, 0000H
NEXT : MOV AL, 00H    ; Get 1st number from the table.
        MOV DH,AL
        MOV CL, 05H    ; Count for display
        MOV DL, E1H    ; Selection code for 1st display
AGAIN : XLAT           ;
        OUT 04H, AL    ; Out the code for the first
                        ; number to port 04H.
        MOV AL,DL      ; Get to be enabled display code.
        OUT 08H,AL    ; Select 1st display.
        ROL DL         ; decide code for selecting next
        INC DH         ; display for next number
        MOV AL,DH      ; get next num. to be displayed.
LOOP AGAIN          ; Repeat five times
JMP NEXT            ; Continue the procedure

```

#### Program 5.4 ALP for Problem 5.9

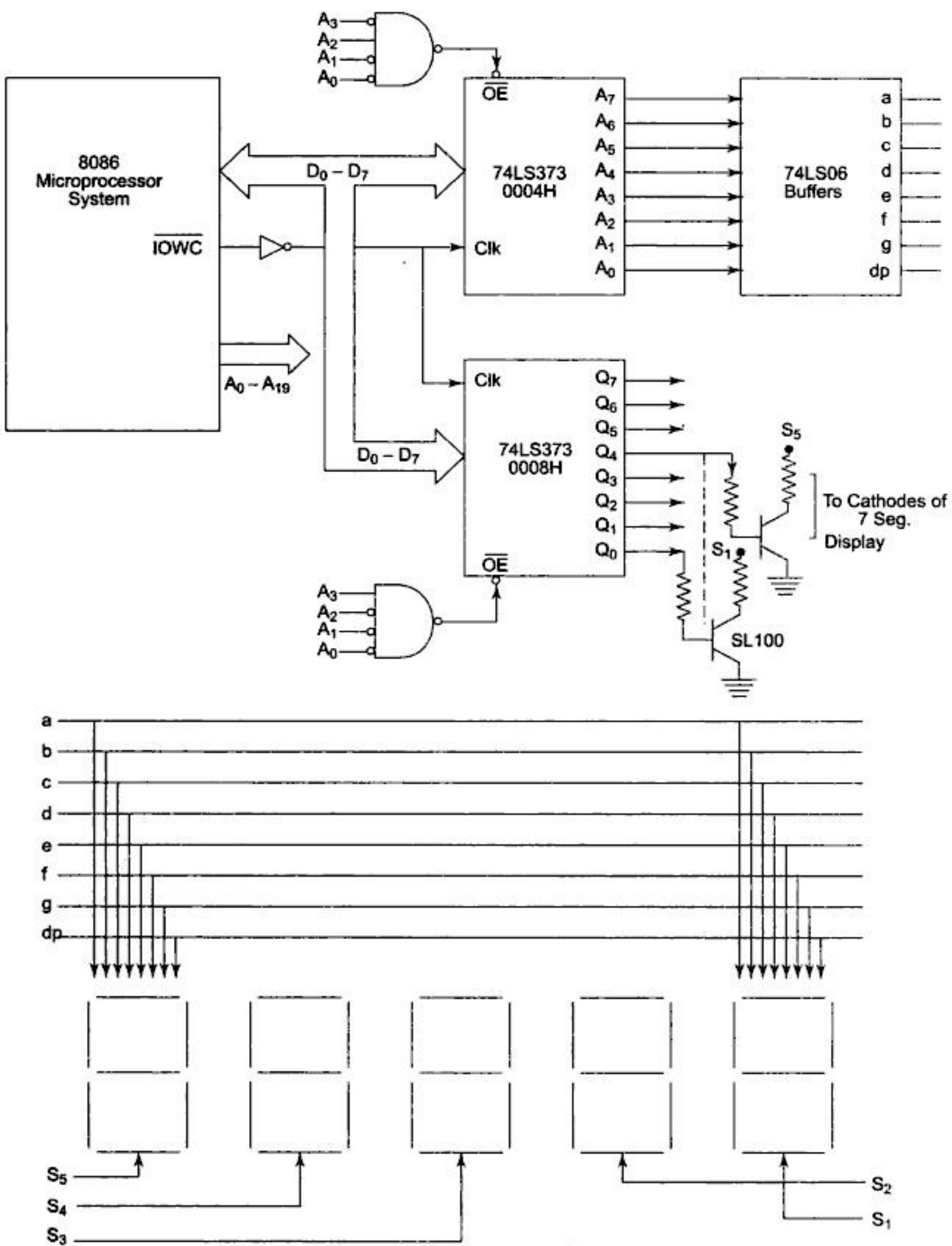
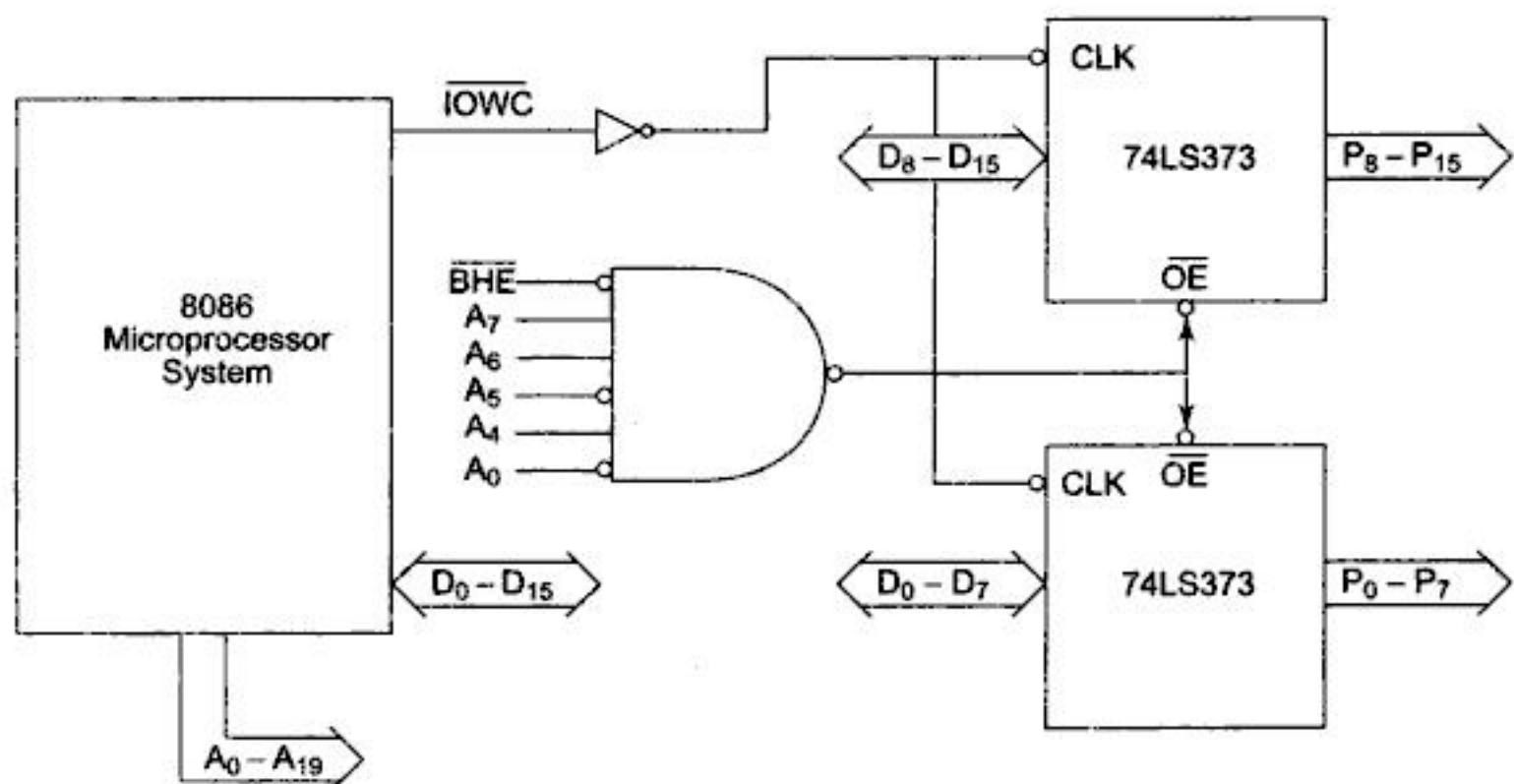


Fig. 5.15 Interfacing Circuit Diagram of Problem 5.9

For interfacing a 16-bit output port with 8086, we may use two 8-bit output ports to form a 16-bit port, with a single address, as shown in Fig. 5.16. Both the 8-bit ports are in this case addressed as a single 16-bit port with a single address.



**Fig. 5.16** Interfacing a Circuit of Problem 5.10

The OUT instruction of 8086 is able to output 16-bit data directly in a single cycle, and the programming technique is identical to that of the 8-bit port. The instruction uses 16-bit register AX as source operand as shown:

```
OUT Port_Add, AX
OUT [DX], AX
```

A 16-bit input port may also be interfaced similarly. Note the use of A<sub>0</sub> and BHE signals in interfacing the 16-bit ports. One may find out address of the output port in Fig. 5.16.

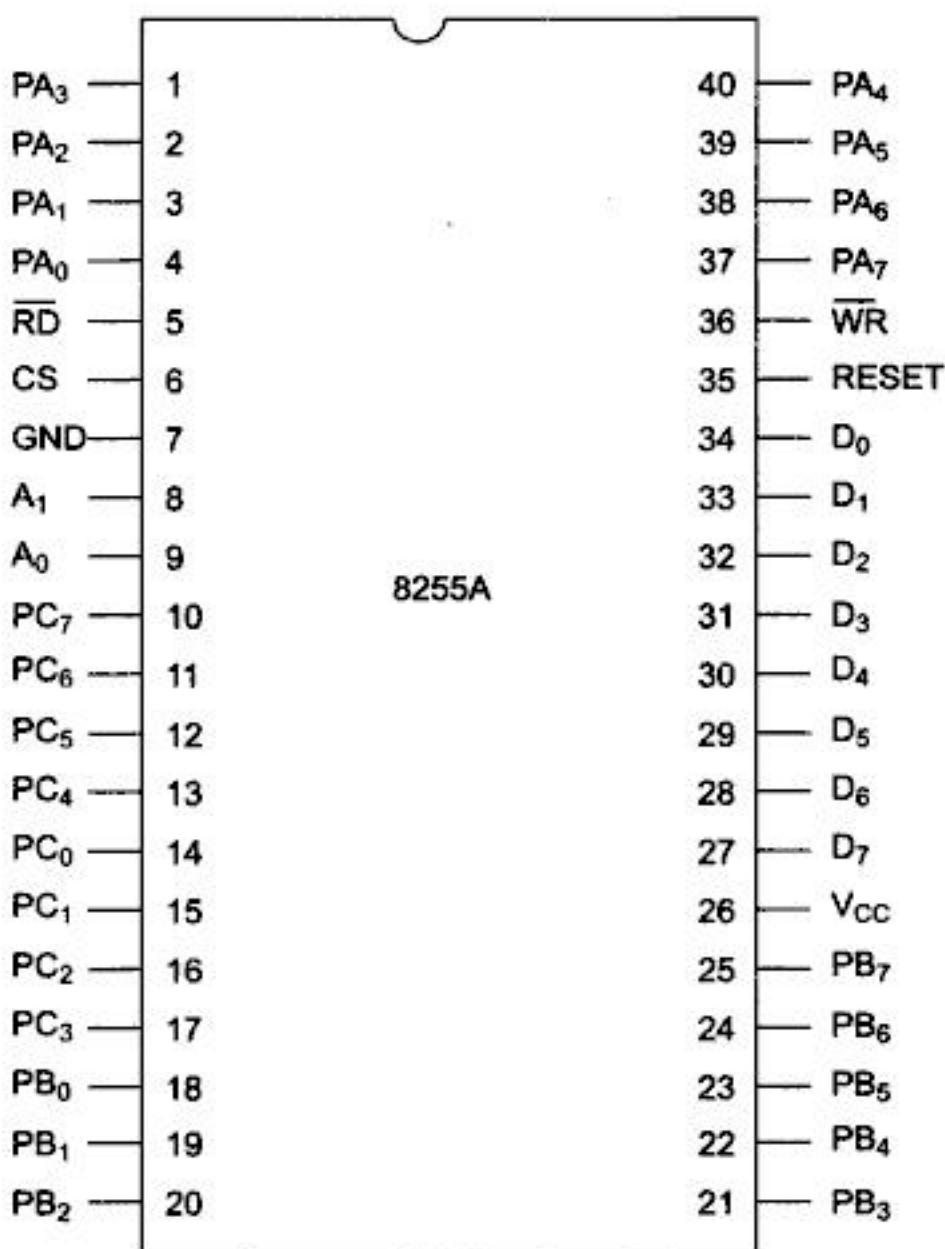
Till now we have studied some common interfacing methods for I/O ports and have also solved some problems based on I/O ports. A few more problems will be discussed while studying the Programmable Peripheral Input/Output port chip 8255. The ports in this section were either input or output, but in 8255 the same port may be programmed either to work as input port or as output port. Hence the chip is called Programmable Input-Output (PIO) device.

#### 5.4 PIO 8255 [PROGRAMMABLE INPUT-OUTPUT PORT]

The parallel input-output port chip 8255 is also known as programmable peripheral input-output port. The Intel's 8255 is designed for use with Intel's 8-bit, 16-bit and higher capability microprocessors. It has 24 input/output lines which may be individually programmed in two groups of twelve lines each, or three groups of eight lines. The two groups of I/O pins are named as Group A and Group B. Each of these two groups contain a subgroup of eight I/O lines called as 8-bit port and another subgroup of four I/O lines or a 4-bit port. Thus Group A contains an 8-bit port A along with a 4-bit port, C upper. The port A lines are identified by symbols PA<sub>0</sub>-PA<sub>7</sub>, while the port C lines are identified as PC<sub>4</sub>-PC<sub>7</sub>. Similarly,



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Fig. 5.17(b) 8255A Pin Configuration**

**PA<sub>7</sub>-PA<sub>0</sub>** These are eight port A lines that act as either latched output or buffered input lines depending upon the control word loaded into the control word register.

**PC<sub>7</sub>-PC<sub>4</sub>** Upper nibble of port C lines. They may act as either output latches or input buffers lines. This port also can be used for generation of handshake lines in mode 1 or mode 2.

**PC<sub>3</sub>-PC<sub>0</sub>** These are the lower port C lines, other details are the same as PC7-PC4 lines.

**PB<sub>0</sub>-PB<sub>7</sub>** These are the eight port B lines which are used as latched output lines or buffered input lines in the same way as port A.

**RD** This is the input line driven by the microprocessor and should be low to indicate read operation to 8255.

**WR** This is an input line driven by the microprocessor. A low on this line indicates write operation.

**CS** This is a chip select line. If this line goes low, it enables the 8255 to respond to RD and WR signals, otherwise RD and WR signals are neglected.

**A<sub>1</sub>-A<sub>0</sub>** These are the address input lines and are driven by the microprocessor. These lines (A<sub>1</sub> – A<sub>0</sub>) with RD, WR and CS form the following operations for 8255. These address lines are used for



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

### 5.5.1 BSR Mode

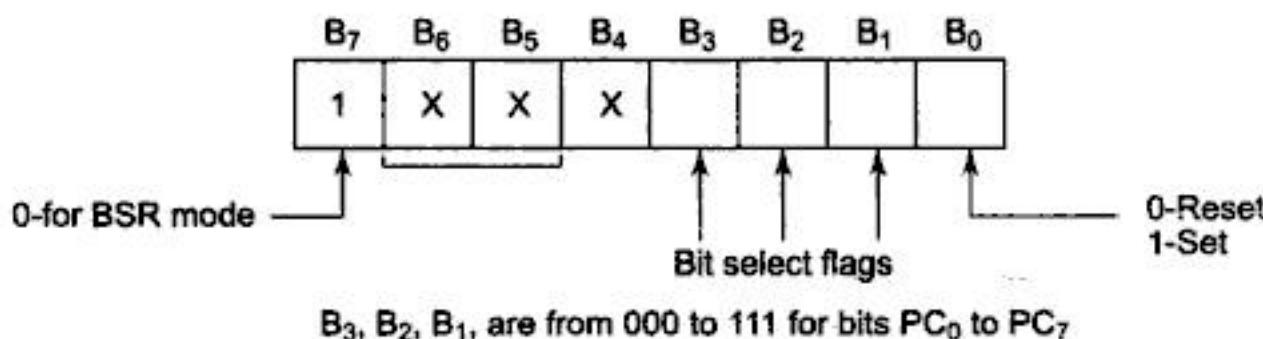
In this mode, any of the 8-bits of port C can be set or reset depending on  $B_0$  of the control word. The bit to be set or reset is selected by bit select flags  $B_3$ ,  $B_2$  and  $B_1$  of the CWR as given in Table 5.10. The CWR format is shown in Fig. 5.18(a).

**Table 5.10**

$B_3$	$B_2$	$B_1$	<i>Selected Bits of port C</i>
0	0	0	$B_0$
0	0	1	$B_1$
0	1	0	$B_2$
0	1	1	$B_3$
1	0	0	$B_4$
1	0	1	$B_5$
1	1	0	$B_6$
1	1	1	$B_7$

### 5.5.2 I/O MODES

**MODE 0 (Basic I/O mode)** This mode is also known as *basic input/output mode*. This mode provides simple input and output capability using each of the three ports. Data can be simply read from and written to the input and output ports respectively, after appropriate initialisation.



**Fig. 5.18(a) BSR Mode Control Word Register Format**

The salient features of this mode are as listed below:

- (i) Two 8-bit ports (port A and port B) and two 4-bit ports (port C upper and lower) are available. The two 4-bit ports can be combinedly used as a third 8-bit port.
- (ii) Any port can be used as an input or output port.
- (iii) Output ports are latched. Input ports are not latched.
- (iv) A maximum of four ports are available so that overall 16 I/O configurations are possible.

All these modes can be selected by programming a register internal to 8255, known as Control Word Register (CWR) which has two formats. The first format is valid for I/O modes of operation, i.e. modes 0, mode 1 and mode 2 while the second format is valid for bit set/reset (BSR) mode of operation. This format is shown in Fig. 5.18(b).

Now let us consider some interfacing problems so as to elaborate the hardware interfacing and I/O programming ideas using 8255 in mode 0.

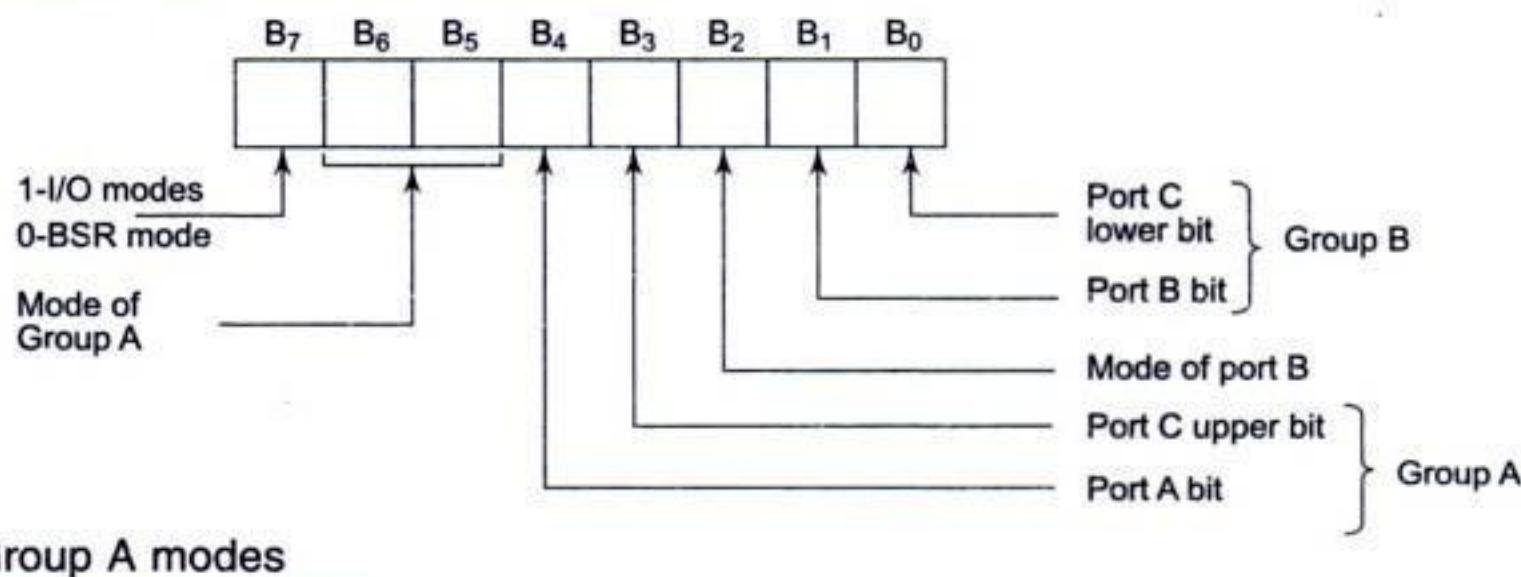
**Problem 5.10**

Interface an 8255 with 8086 to work as an I/O port. Initialize port A as output port, port B as input port and port C as output port. Port A address should be 0740H. Write a program to sense switch positions  $SW_0-SW_7$  connected at port B. The sensed pattern is to be displayed on port A, to which 8 LEDs are connected, while the port C lower displays number of on switches out of the total eight switches.

**Solution** The control word is decided upon as follows:

B7	B6	B5	B4	B3	B2	B1	B0	Control word
1	0	0	0	0	0	1	0	= 82H
I/O mode	Port A		Port	Port	Port	Port	Port	
in mode 0		A,o/p	C,o/p	B,mode 0	B,i/p	C,o/p		

Thus 82H is the control word for the requirements in the problem. The port address decoding can be done as given below. The 8255 is to be interfaced with lower order data bus, i.e.  $D_0-D_7$ . The  $A_0$  and  $A_1$  pins of 8255 are connected to  $A_{01}$  and  $A_{02}$  pins of the microprocessor respectively. The  $A_{00}$  pin of the microprocessor is used for selecting the transfer on the lower byte of the data bus. Hence any change in the status of  $A_{00}$  does not affect the port to be selected, rather  $A_{01}$  and  $A_{02}$  of the microprocessor decide the port to be selected as they are connected to  $A_0$  and  $A_1$  of 8255. The 8255 port addresses are tabulated as shown below.



B <sub>6</sub>	B <sub>5</sub>	Mode
0	0	mode 0
0	1	mode 1
1	0	mode 2
1	1	x

- (i) Port B mode is either 0 or 1 depending upon B2 bit.
- (ii) A port is an output port if the port bit is 0 else it is input port

**Fig. 5.18(b) I/O Mode Control Word Register Format**

8255 Ports	I/O Address lines														Hex. Port Addresses	
	$A_{15}$	$A_{14}$	$A_{13}$	$A_{12}$	$A_{11}$	$A_{10}$	$A_{09}$	$A_{08}$	$A_{07}$	$A_{06}$	$A_{05}$	$A_{04}$	$A_{03}$	$A_{02}$	$A_{01}$	$A_{00}$
Port A	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0740H
Port B	0	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0742H

(Contd.)

(Contd.)

I/O Address lines														Hex. Port Addresses			
Ports	$A_{15}$	$A_{14}$	$A_{13}$	$A_{12}$	$A_{11}$	$A_{10}$	$A_{09}$	$A_{08}$	$A_{07}$	$A_{06}$	$A_{05}$	$A_{04}$	$A_{03}$	$A_{02}$	$A_{01}$	$A_{00}$	
Port C	0	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0	0744H
CWR	0	0	0	0	0	1	1	1	0	1	0	0	0	1	1	0	0746H

Let us use absolute decoding scheme that uses all the 16 address lines for deriving the device address pulse. Out of  $A_0 - A_{15}$  lines, two address lines  $A_{02}$  and  $A_{01}$  are directly required by 8255 for the three port and CWR address decoding. Hence only  $A_3$  to  $A_{15}$  are used for decoding addresses. The complete hardware scheme is shown in Fig. 5.19. In the diagram, the 8086 is assumed to be in the maximum mode so that  $\overline{\text{IORD}}$  and  $\overline{\text{IOWR}}$  are readily available. If the 8086 is in minimum mode,  $\overline{\text{RD}}$  and  $\overline{\text{WR}}$  of 8086 are to be connected accordingly to 8255 and  $\text{M}/\overline{\text{IO}}$  pin is combined with the chip select of above hardware suitably so as to select the 8255 when  $\text{M}/\overline{\text{IO}}$  is low.

The ALP for the problem is developed as follows:

```

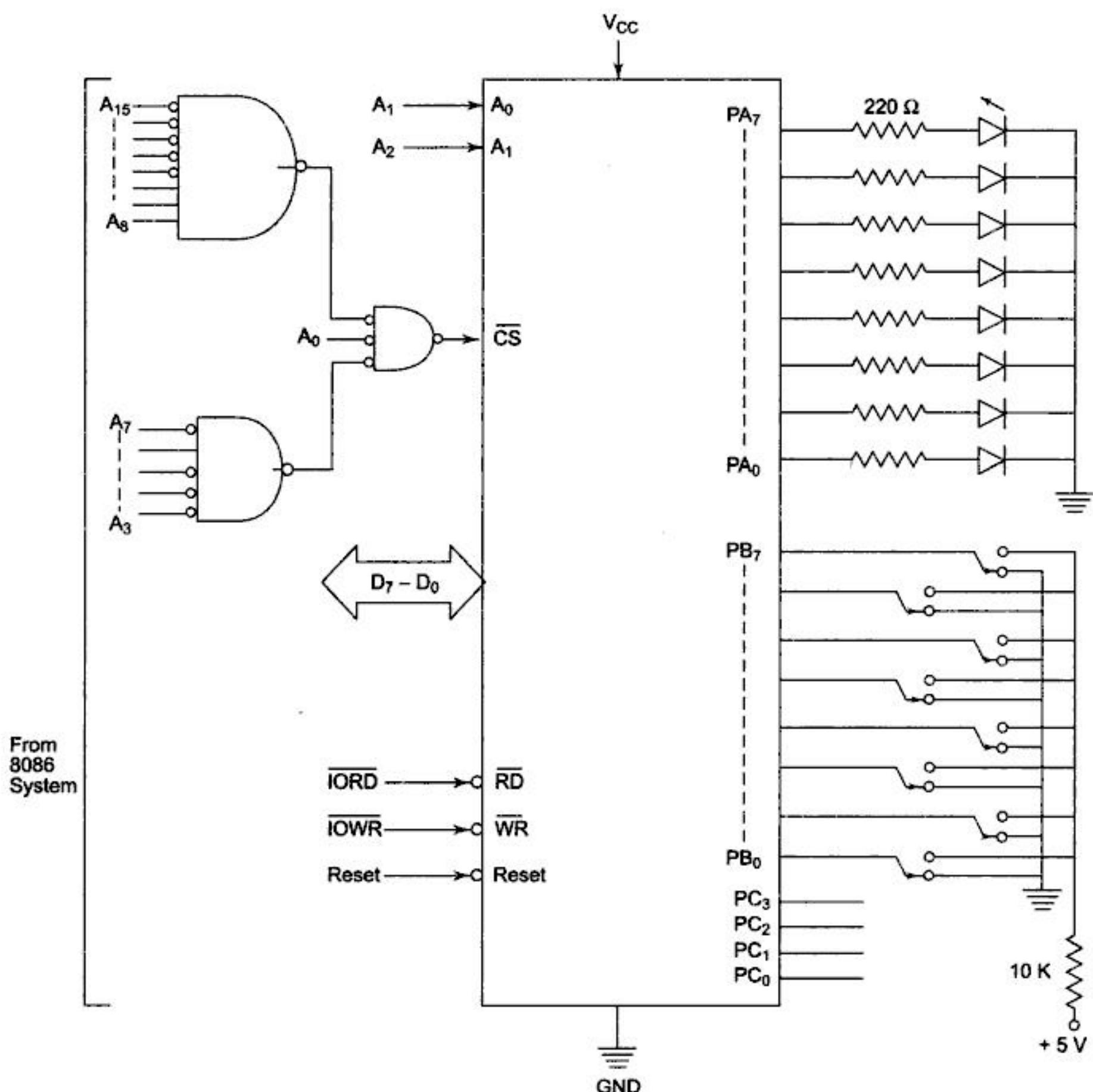
MOV DX, 0746 H ; Initialise CWR with
MOV AL, 82 H    ; control word 82H
OUT DX, AL      ;
SUB DX,04       ; Get address of port B in DX
IN AL, DX       ; Read port B for switch
SUB DX,02       ; positions in to AL and get port A address
                ; in DX.
OUT DX, AL      ; Display switch positions on port A
MOV BL, 00 H    ; Initialise BL for switch count
MOV CH, 08H    ; Initialise CH for total switch number
YY: ROL AL      ; Rotate AL through carry to check,
                ; whether the switches are on or
INC BL          ; off, i.e. either 1 or 0
XX : DEC CH      ; Check for next switch. If
                ; all switch are checked, the
JNZ YY          ; number of on switches are
MOV AL, BL      ; in BL. Display it on port C
ADD DX, 04      ;
OUT DX,AL       ; lower.
HLT             ; Stop

```

#### Program 5.5 ALP for Problem 5.10

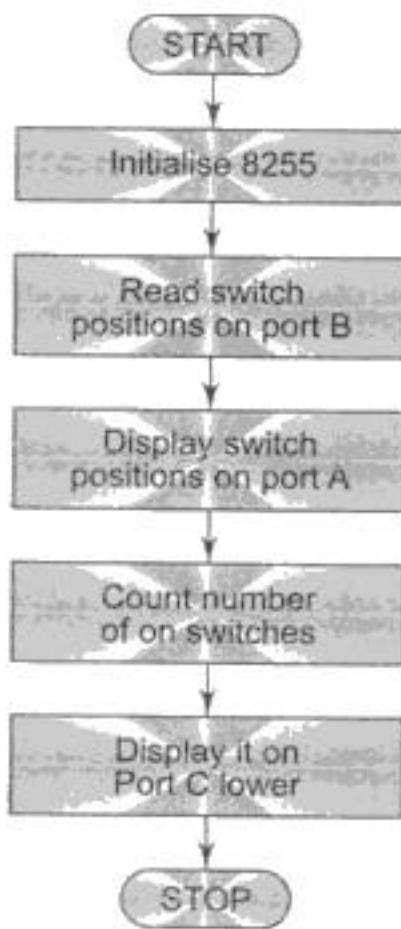
#### Problem 5.11

Interface a 4\*4 Keyboard with 8086 using 8255. and write an ALP for detecting a key closure and return the key code in AL. The debouncing period for a key is 10 ms. Use software key debouncing technique. DEBOUNCE is an available 10 ms delay routine.

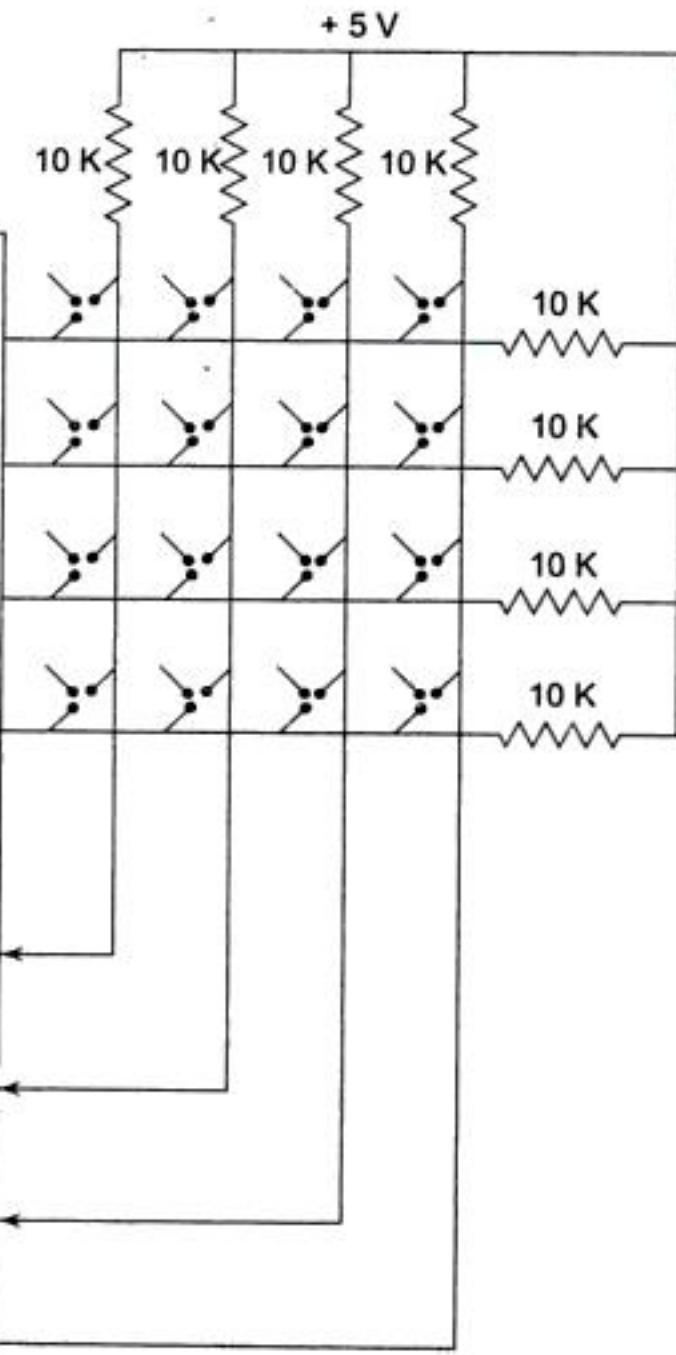
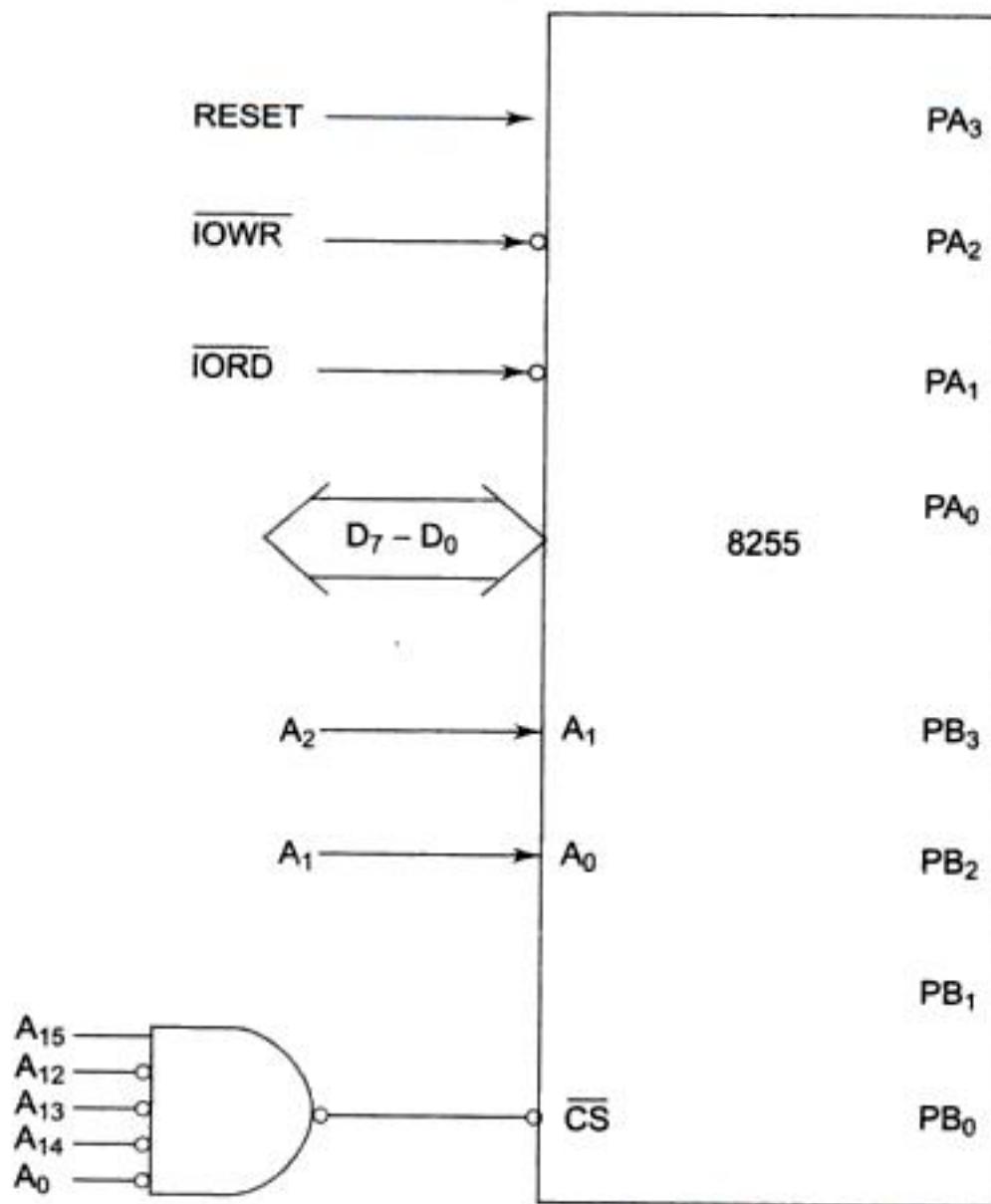


**Fig. 5.19** 8255 Interfacing with 8086 for Problem 5.10

**Solution** Port A is used as output port for selecting a row of keys while port B is used as an input port for sensing a closed key. Thus the keyboard lines are selected one by one through port A and the port B lines are polled continuously till a key closure is sensed. Then routine DEBOUNCE is called for key debouncing. The key code is decided depending upon the selected row and a low sensed column. The hardware circuit diagram is shown in Fig. 5.21.



**Fig. 5.20** Flow Chart for the ALP of Problem 5.10



**Fig. 5.21** Interfacing  $4 \times 4$  Keyboard for Problem 5.12



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

hence only one segment is used for storing the program code, i.e. code segment (CS). This program is written in MASM syntax. The 8255 is again interfaced to the lower byte of the 8086 data bus. Absolute decoding scheme is not used here to implement the circuit using minimum hardware.

```

CODE SEGMENT
ASSUME CS : CODE
START: MOV AL, 82H      ; Load CWR with
        MOV DX, 8006H    ; control word
        OUT DX, AL       ; required
        MOV BL, 00H       ; Initialize BL for key code
        XOR AX, AX       ; Clear all flags
        MOV DX, 8000H    ; Port Address in AX.
        OUT DX, AL       ; Ground all rows.
        ADD DX, 02       ; Port B address in DX.
        WAIT : IN AL, DX  ; Read all columns.
        AND AL, 0F H     ; Mask data lines D7-D4.
        CMP AL, 0F H     ; Any key closed?
        JZ WAIT          ; If not, wait till key
        CALL DEBOUNCE   ; closure else wait for 10 ms
        MOV AL, 7FH       ; Load data byte to ground
        MOV BH, 04H       ; a row and set row counter.
NXTROW : ROL AL, 01    ; Rotate AL to ground next row.
        MOV CH, AL       ; Save data byte to ground next row.
        SUB DX, 02       ; Output port address is in DX.
        OUT DX, AL       ; Ground one of the rows.
        ADD DX, 02       ; Input port address is in DX.
        IN AL, DX        ; Read input port for key closure.
        AND AL, 0FH       ; Mask D4-D7.
        MOV CL, 04H       ; Set column counter.
NXTCOL : ROR AL, 01    ; Move D0 in CF.
        JNC CODEKY      ; Key closure is found, if CF=0.
        INC BL           ; Increment BL for next binary
        key code.
        DEC CL           ; Decrement column counter,
        ; if no key closure found.
        JNZ NXTCOL      ; Check for key closure in next column
        MOV AL, CH       ; Load data byte to ground next row.
        DEC BH           ; if no key closer found in column
        ; get ready to ground next row.
        JNZ NXTROW      ; Go back to ground next row.
        JMP WAIT         ; Jump back to check for key.
        ; closure again.
CODEKY : MOV AL, BL    ; Key code is transferred to AL.
        MOV AH, 4CH       ; Return to DOS prompt.
        INT 21 H

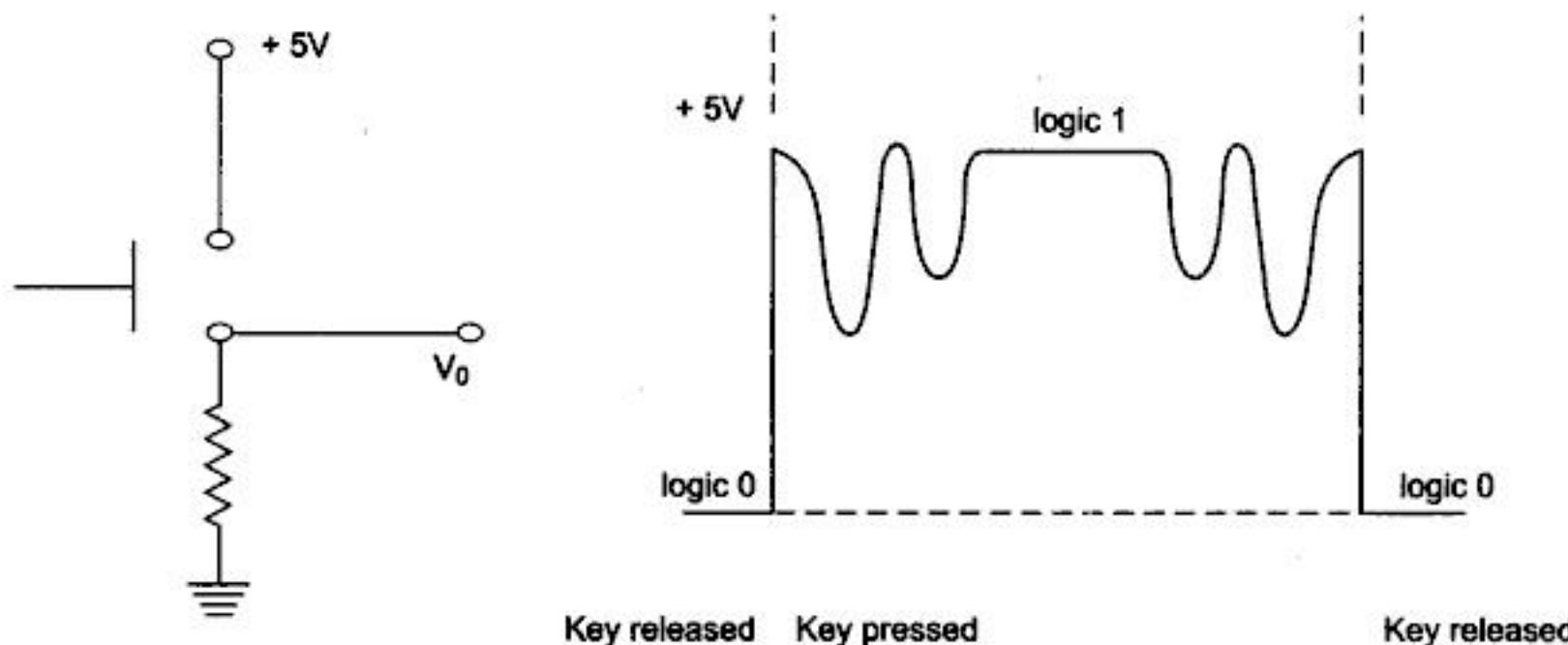
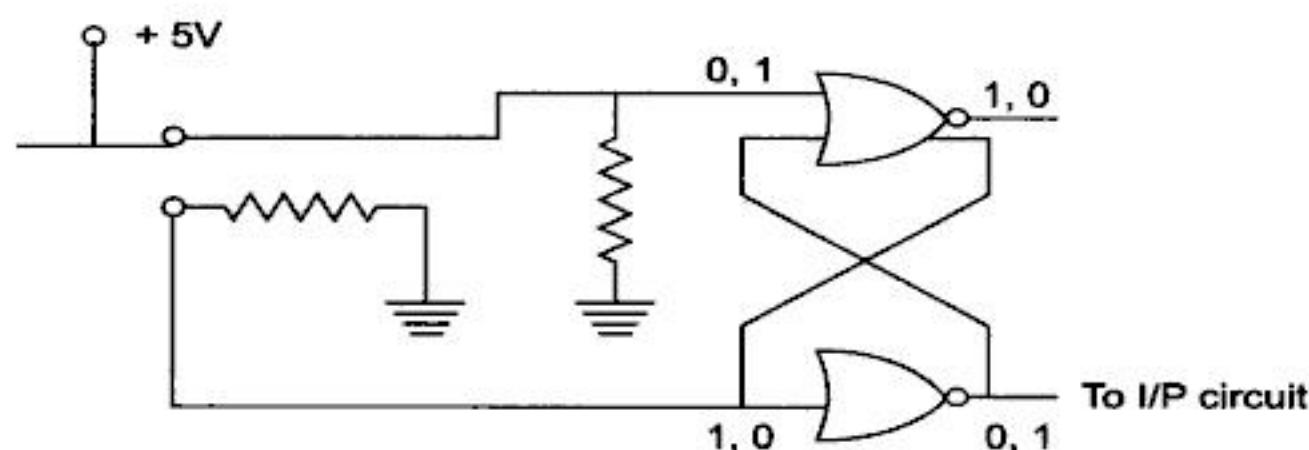
```

This procedure generates 10 ms delay at 5 MHz operating frequency.

```

DEBOUNCE PROC NEAR
    MOV CL, 0E2H
BACK:   NOP
        DEC CL
        JNZ BACK
        RET
DEBOUNCE ENDP
CODE      ENDS
END START

```

**Program 5.6 ALP for Problem 5.11****Fig. 5.23 A Mechanical Key and Its Response****Fig. 5.24 Hardware Debouncing Circuit**

**Key Debounce** Whenever a mechanical push-button is pressed or released once, the mechanical components of the key do not change the position smoothly, rather, it generates a transient response as shown in Fig. 5.23. These transient variations may be interpreted as the multiple key pressures and responded accordingly by the microprocessor system. To avoid this problem, two schemes are suggested: the first one utilizes a bistable multivibrator at the output of the key to debounce it as shown in Fig. 5.24. The other scheme suggests that the microprocessor should be made to wait for the transient

period (usually 10 msec), so that the transient response settles down and reaches a steady state. A logic '0' will be read by the microprocessor when the key is pressed.

In a number of high precision applications, a designer may need to read or write more than 8-bits of data. In these cases, a system designer may have two options—the first is to have more than one 8-bit port, read(write) the ports one by one and then form the multibyte data; the second option allows forming 16-bit ports using two 8-bit ports and use 16-bit read or write operations. The following example elaborates interfacing of a 16-bit port using two 8-bit ports.

### Problem 5.12

Interface 16-bit 8255 ports with 8086. The address of port A is F0H.

**Solution** To implement a 16-bit port two 8255s are required. One will act as the lower 8-bit port, i.e. D<sub>0</sub>-D<sub>7</sub>, while the other will act as the upper 8-bit port D<sub>8</sub>-D<sub>15</sub>. The overall scheme is as shown in Fig. 5.25. While initialising AL and AH (AX) both should be loaded with a suitable (common) control word. In this system, port A, port B and port C all may work as 16-bit ports.

### Problem 5.13

Interface an 8255 with 8086 at 80H as an I/O address of port A. Interface five 7 segment displays with the 8255. Write a sequence of instructions to display 1, 2, 3, 4 and 5 over the five displays continuously as per their positions starting with 1 at the least significant position.

**Solution** The hardware scheme for the above problem is shown in Fig. 5.26. In this scheme, I/O port A is multiplexed to carry data for all the 7-segment displays. The port B selects (grounds) one of the displays at a time.

The displays used in the above hardware scheme are common cathode type. To glow a segment, logic 1 is applied on the corresponding line and the corresponding 7-segment display is selected by applying logic 1 on the port line that drives a transistor to ground the common cathode pin of the display. Thus the codes are decided as shown. For a common cathode display, a '1', applied to a segment glows it and a '0' blanks it.

**Table 5.11**

Number to be displayed	PA-dp	PA <sub>0</sub>	PA <sub>1</sub>	PA <sub>2</sub>	PA <sub>3</sub>	PA <sub>4</sub>	PA <sub>5</sub>	PA <sub>6</sub>	PA <sub>7</sub>	Code
1	1	1	0	0	1	1	1	1	1	CF
2	1	0	0	1	0	0	1	0	0	92
3	1	0	0	0	0	1	1	0	0	86
4	1	1	0	0	1	1	0	0	0	CC
5	1	0	1	0	0	1	0	0	0	A4

All these codes, decided as above, are stored in a look up table starting at 2000:0001. The ALP along with comments is given as follows:

```
AGAIN:    MOV CL, 05H      ; Count for displays
          MOV BX, 2000H    ; Initialise data segment
          MOV DS, BX       ; for look up table
```

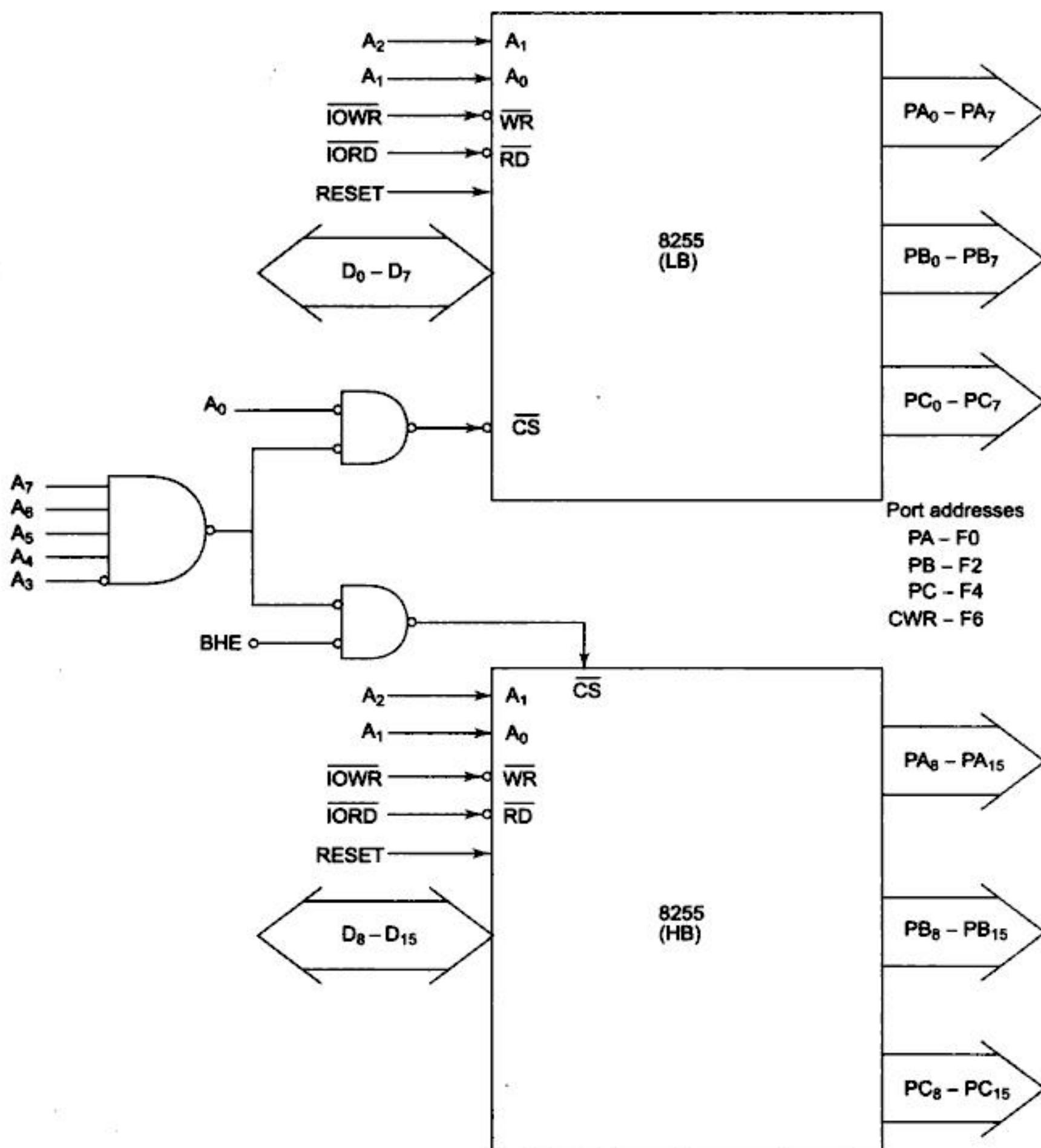


Fig. 5.25 Interfacing 16-bit 8255 Ports with 8086

```

MOV CH, 01H ; 1st number to be displayed
MOV AL, 80H ; Load control word in the
OUT 86H,AL ; CWR
MOV DL,01H ; Enable code for Least significant
            ; 7-seg display
    
```

```

NXTDGT : MOV BX, 0000H ; Set pointer to look up table
      MOV AL, CH ; First no to display
      ; Store number to be displayed in AL.
      XLAT ; Find code from look up table
      OUT 80H,AL ; Display the code
      MOV AL, DL ; Enable the display
      OUT 81H,AL ;
      ; Go for selecting the next display
      ROL DL ; Next number to display
      INC CH ; Decrement count.
      DEC CL ; Go for next digit display
      JNZ NXTDGT ; Repeat the procedure
      JMP AGAIN

```

Program 5.7 ALP for Problem 5.13

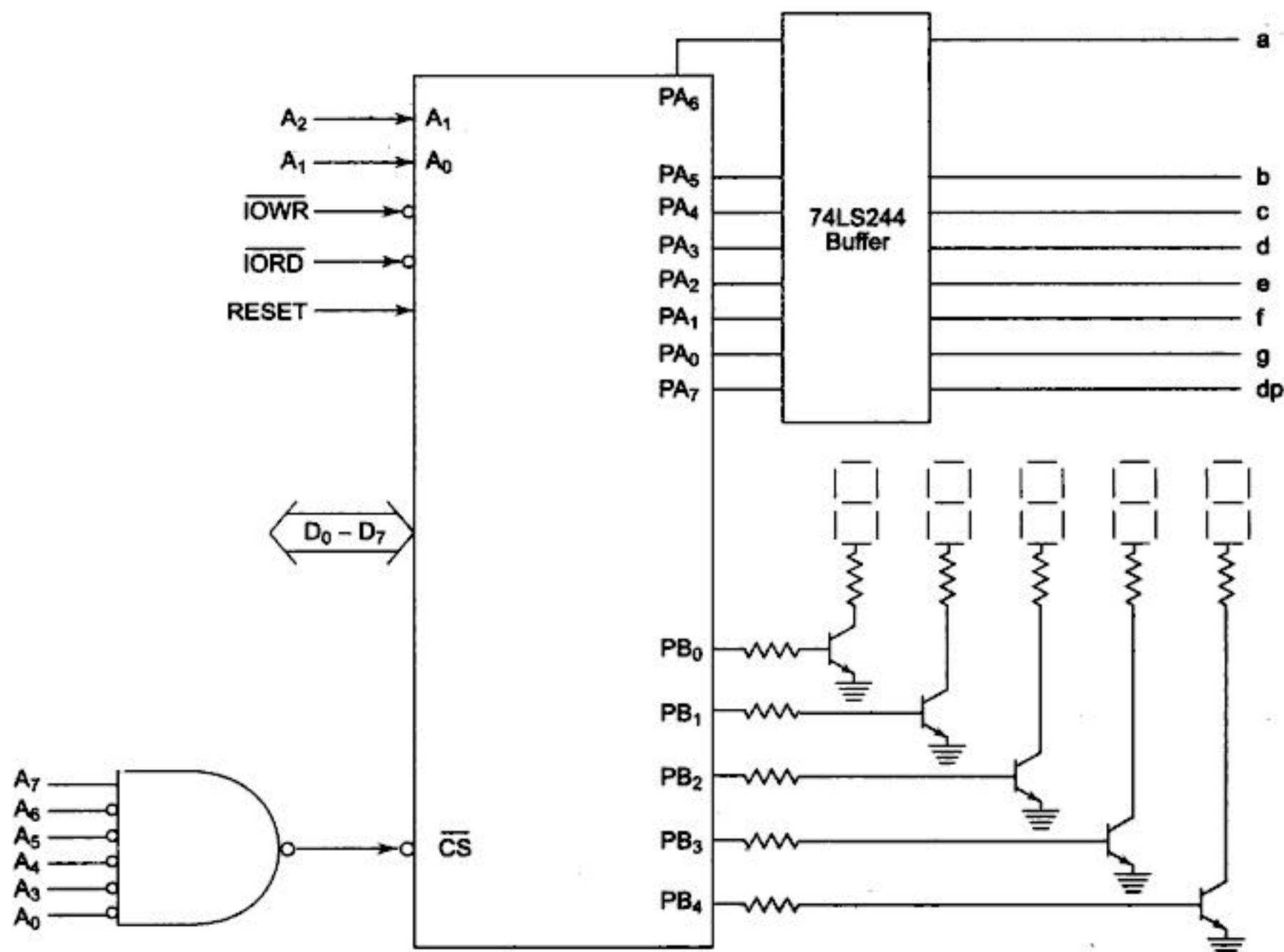


Fig. 5.26 Interfacing Multiplexed 7-Segment Display

**MODE I (Strobed I/O mode)** This mode is also called as strobed input/output mode. In this mode the handshaking signals control the input or output action of the specified port. Port C lines PC<sub>0</sub>-PC<sub>2</sub>, provide strobe or handshake lines for port B. This group which includes port B and PC<sub>0</sub>-PC<sub>2</sub> is called as group B for strobed data input/output. Port C lines PC<sub>3</sub>-PC<sub>5</sub> provide strobe lines for port A. This group including port A and PC<sub>3</sub>-PC<sub>5</sub> forms group A. Thus port C is utilized for generating handshake signals. The salient features of mode 1 are listed as follows:

- (i) Two groups—group A and group B are available for strobed data transfer.
- (ii) Each group contains one 8-bit data I/O port and one 4-bit control/data port.
- (iii) The 8-bit data port can be either used as input or an output port. Both the inputs and outputs are latched.
- (iv) Out of 8-bit port C, PC<sub>0</sub>-PC<sub>2</sub> are used to generate control signals for port B and PC<sub>3</sub>-PC<sub>5</sub> are used to generate control signals for port A. The lines PC<sub>6</sub>, PC<sub>7</sub> may be used as independent data lines.

The control signals for both the groups in input and output modes are explained as follows:

#### Input control signal definitions (mode 1)

**STB (Strobe input)**—If this line falls to logic low level, the data available at 8-bit input port is loaded into input latches.

**IBF (Input buffer full)**—If this signal rises to logic 1, it indicates that data has been loaded into the latches, i.e. it works as an acknowledgement. IBF is set by a low on **STB** and is reset by the rising edge of **RD** input.

**INTR (Interrupt request)** This active high output signal can be used to interrupt the CPU whenever an input device requests the service. INTR is set by a high at **STB** pin and a high at IBF pin. INTE is an internal flag that can be controlled by the bit set/reset mode of either PC<sub>4</sub> (INTE<sub>A</sub>) or PC<sub>2</sub> (INTE<sub>B</sub>) as shown in Figs 5.27(a) and (b). INTR is reset by a falling edge on **RD** input. Thus an external input device can request the service of the processor by putting the data on the bus and sending the strobe signal. Figure 5.27 explains the signal definitions clearly.

The strobed data input cycle waveforms are shown in Fig. 5.28(a).

#### Output control signal definitions (mode 1)

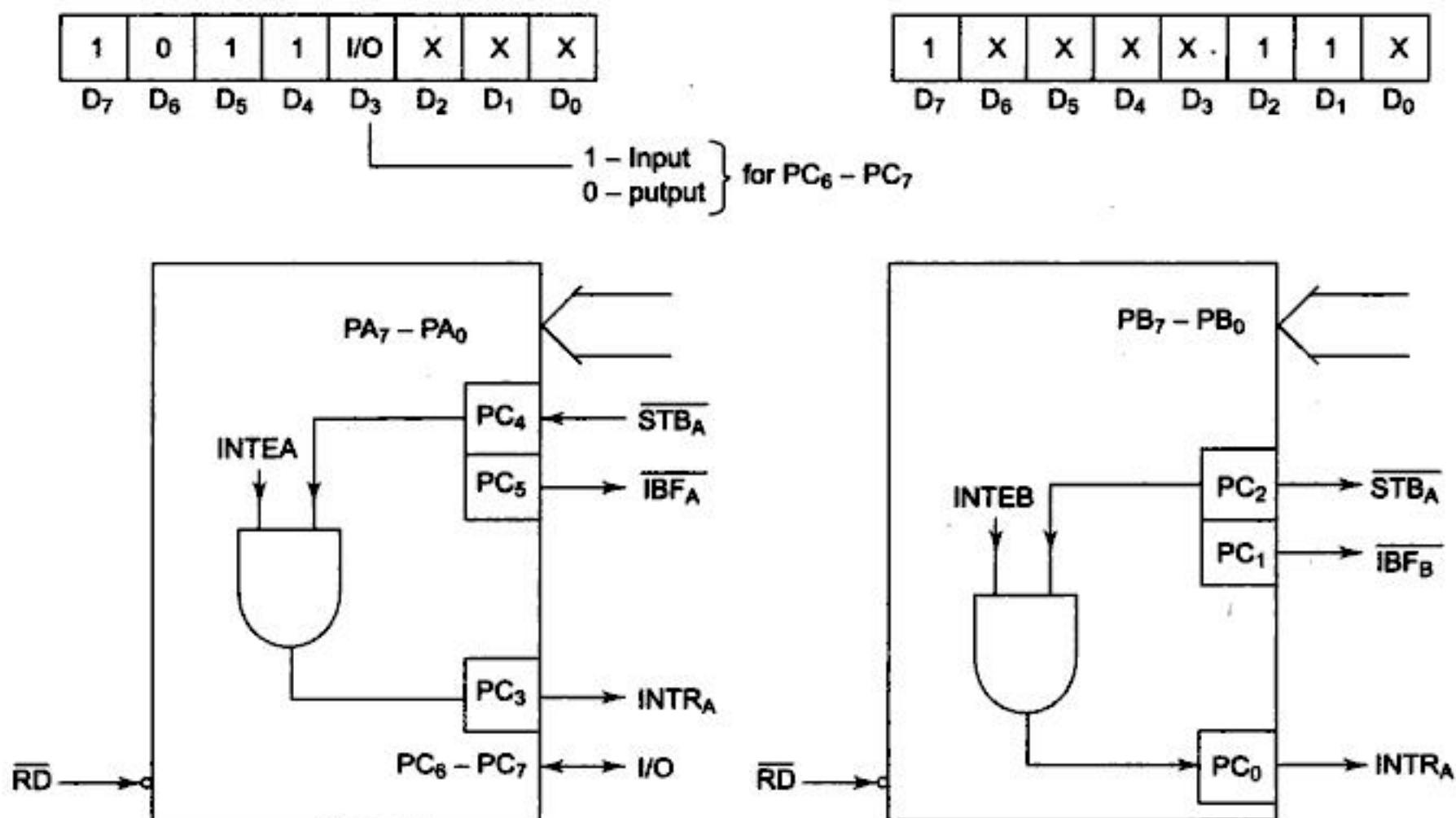
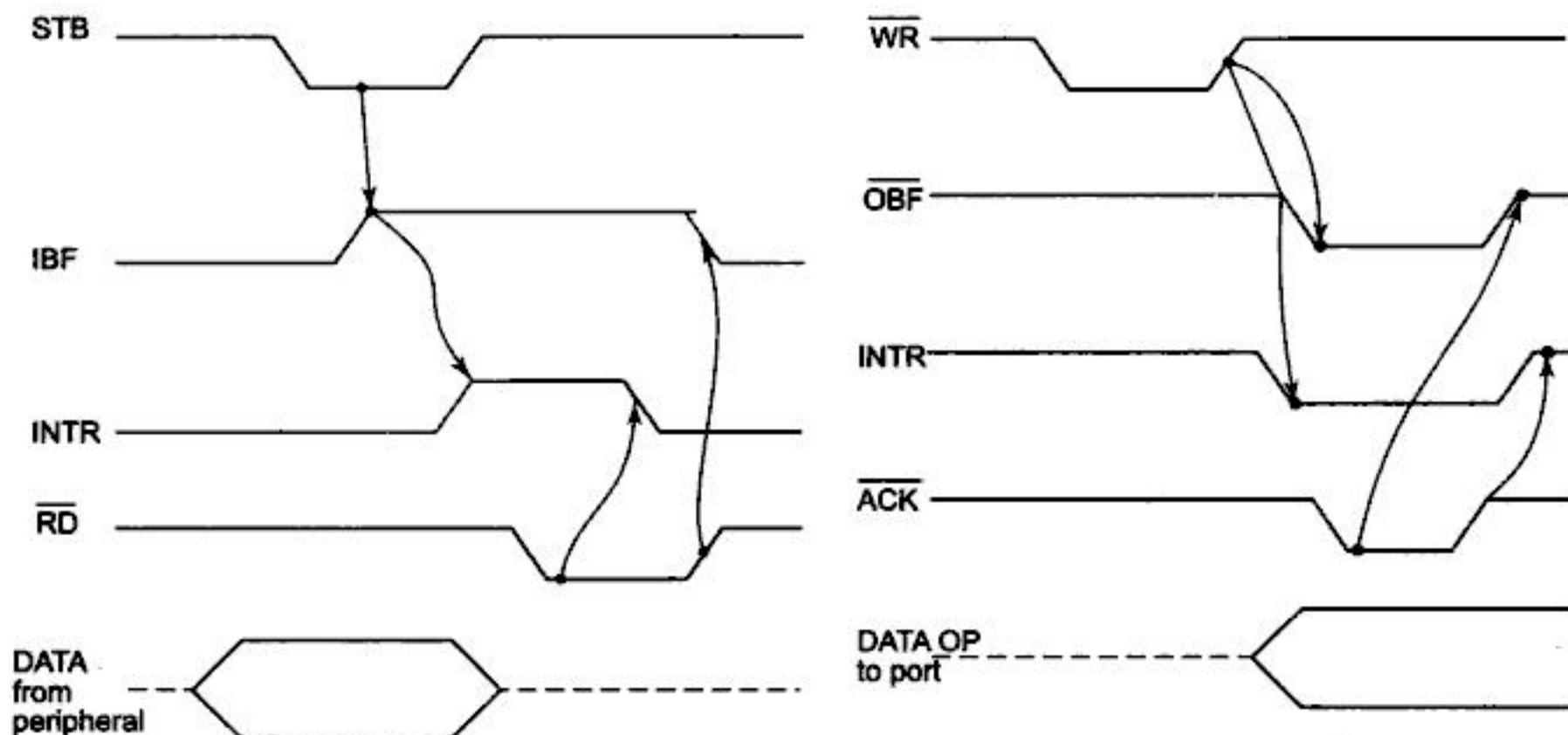
**OBF (Output buffer full)**—This status signal, whenever falls to logic low, indicates that the CPU has written data to the specified output port. The OBF flip-flop will be set by a rising edge of **WR** signal and reset by a low going edge at the **ACK** input.

**ACK (Acknowledge input)**—**ACK** signal acts as an acknowledgement to be given by an output device. **ACK** signal, whenever low, informs the CPU that the data transferred by the CPU to the output device through the port is received by the output device.

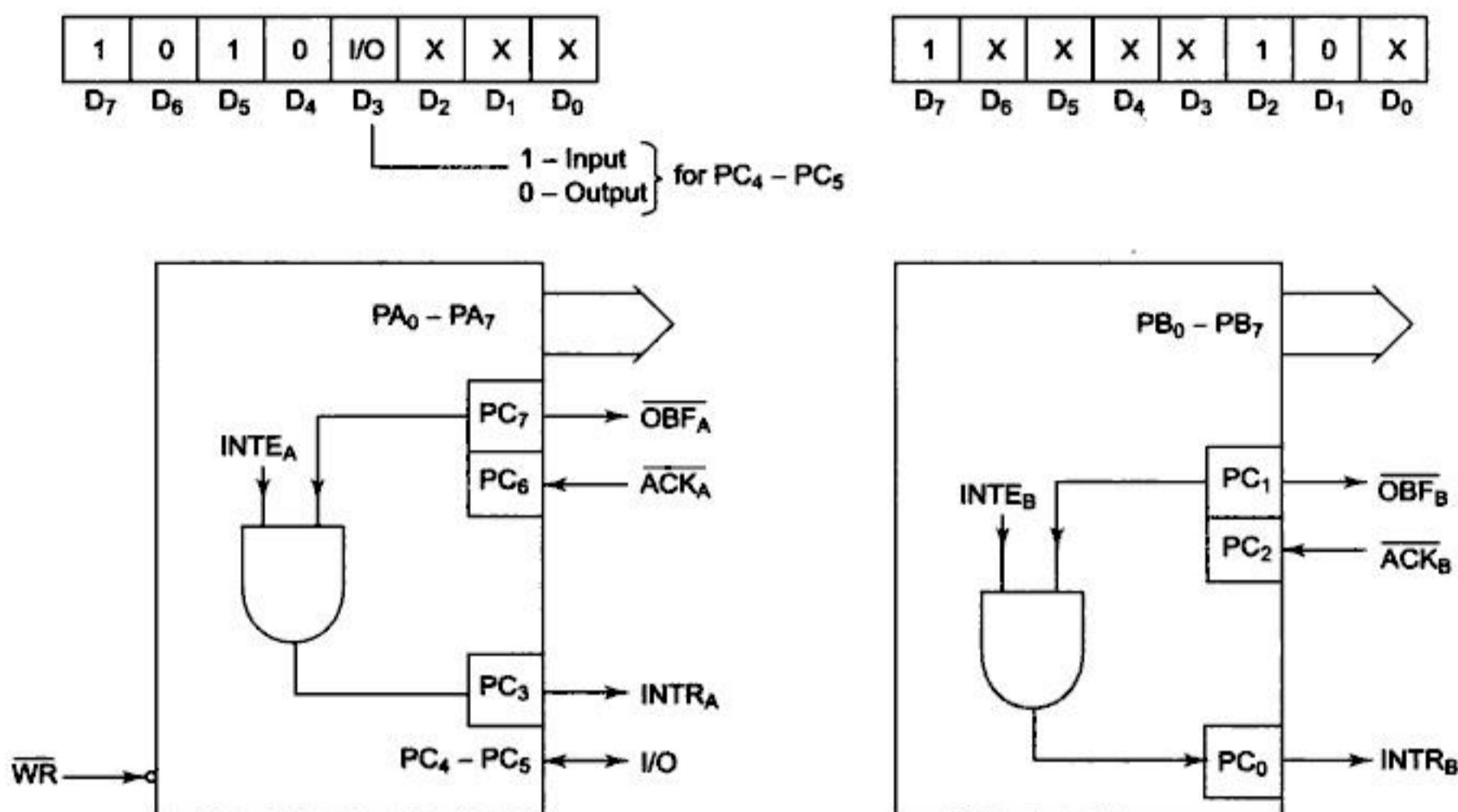
**INTR (Interrupt request)**—Thus an output signal that can be used to interrupt the CPU when an output device acknowledges the data received from the CPU. INTR is set when **ACK**, **OBF** and INTE are '1'. It is reset by a falling edge on **WR** input. The INTEA and INTEB flags are controlled by the bit set-reset mode of PC<sub>6</sub> and PC<sub>2</sub>, respectively.

The waveforms in Fig. 5.28 may help in understanding the handshake data transfers. The following Figs 5.29 (a) and (b) explains the signal definitions of mode 1 in output mode clearly.

## Input control signal definitions in Mode 1

**Fig. 5.27 (a) Mode 1 Control Word Group A I/P (b) Mode 1 Control Word Group B I/P****Fig. 5.28 (a) Mode 1 Strobed Input Data Transfer (b) Mode 1 Strobed Data Output**

Output control signal definitions Mode 1

**Fig. 5.29 (a) Mode 1 Control Word Group A (b) Mode 1 Control Word Group B**

After discussing mode 1 in necessary details, let us now consider some hardware interfacing examples that utilize this mode of operation of 8255.

#### Problem 5.14

Interface a standard IEEE-488 parallel bus printer with 8086. Draw the necessary hardware scheme required for the same and write an ALP to print a character whose ASCII code is available in AL.

**Solution** Before going through this solution, one should refer to the standard Centronics, INB or EPSON printer pin configuration, given in Table 5.12. There are two types of parallel cables used to connect a microcomputer with a printer, viz. 25 pin cables and 36 pin cables. Basically the 25 pin and the 36 pin cables are similar except for the 11 extra pins for ground (GND) used as 'RETURN' lines for different signals.

The group A is used in mode 1 for handshake data transfer so that port A is used for data transfer and port C lines PC<sub>3</sub>-PC<sub>5</sub> are used as handshake lines. Port B lines are used for checking the printer status, hence port B is used as input port in mode 0. Port C lower is used as output port for enabling the printer. The control words are shown in Fig. 5.30.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

<i>Signal Pin No.</i>	<i>Return Pin No.</i>	<i>Signal Pin No.</i>	<i>Direction</i>	<i>Description</i>
31	—	<u>INIT</u>	IN	When the level of this signal becomes "low" the printer controller is reset to its initial state and the print buffer is cleared. This signal is normally at "high" level, and its pulse width must be more than 50 $\mu$ s at the receiving terminal.
32	—	<u>ERROR</u>	OUT	The level of this signal becomes "low" when the printer is in "Paper End" state, "Offline" state and "Error" state.
33	—	GND	—	Same as with pin numbers 19 to 30.
34	—	NC	—	Not used.
35	—	—	—	Pulled up to + 5 Vdc through 4.7 k-ohms resistance.
36	—	<u>SLCT IN</u>	IN	Data entry to the printer is possible only when the level of this signal is "low". (Internal fixing can be carried out with DIP SW 1-8. The condition at the time of shipment is set "low" for this signal.)

- Note:**
1. "Direction" refers to the direction of signal flow as viewed from the printer.
  2. "Return" denotes "Twisted-Pair Return" and is to be connected at signal-ground level. When wiring the interface, be sure to use a twisted-pair cable for each signal and never fail to complete connection on the return side. To prevent noise effectively, these cables should be shielded and connected to the chassis of the system unit.
  3. All interface conditions are based on TTL, level. Both the rise and fall times of each signal must be less than 0.2  $\mu$ s.
  4. Data transfer must not be carried out by ignoring the ACKNLG or BUSY signal. (Data transfer to this printer can be carried out only after interfacing the ACKNLG signal or when the level of the BUSY signal is "low".)
  5. Remaining pins on the connector are no connection pins.
  6. x-not available in 25 pins connector.

**Printer Operation** The printer interface connections with 8255 and the printer connector shown in Fig. 5.31 and Fig. 5.32 respectively. First of all the printer should be initialised by sending a 50  $\mu$ s (minimum) pulse on the INIT pin of the printer. Then the BUSY pin is to be checked to confirm if the printer is ready. If this signal is low, it indicates that the printer is ready to accept a character from the CPU. Port pins of 8255 may not have sufficient drive capacity to drive the printer input signals so that the open collector buffers 74LSOY are used to enhance the drive capacity. When this happens the ASCII code of the character to be printed is sent on the eight parallel port lines. Once the data is sent on eight parallel lines, the STROBE signal is activated after at least 0.5  $\mu$ s, to indicate that the data is available on the eight data lines. The falling edge of the STROBE signal causes the printer to make its BUSY pin high, indicating that the printer is busy. After a minimum period of 0.5  $\mu$ s, the STROBE signal can be sent high. The data must be valid on the data lines for at least 0.5  $\mu$ s after the STROBE signal goes high. After receiving the appropriate STROBE pulse, the printer starts the necessary electromechanical action to print the character and when it is ready to receive the next character, it

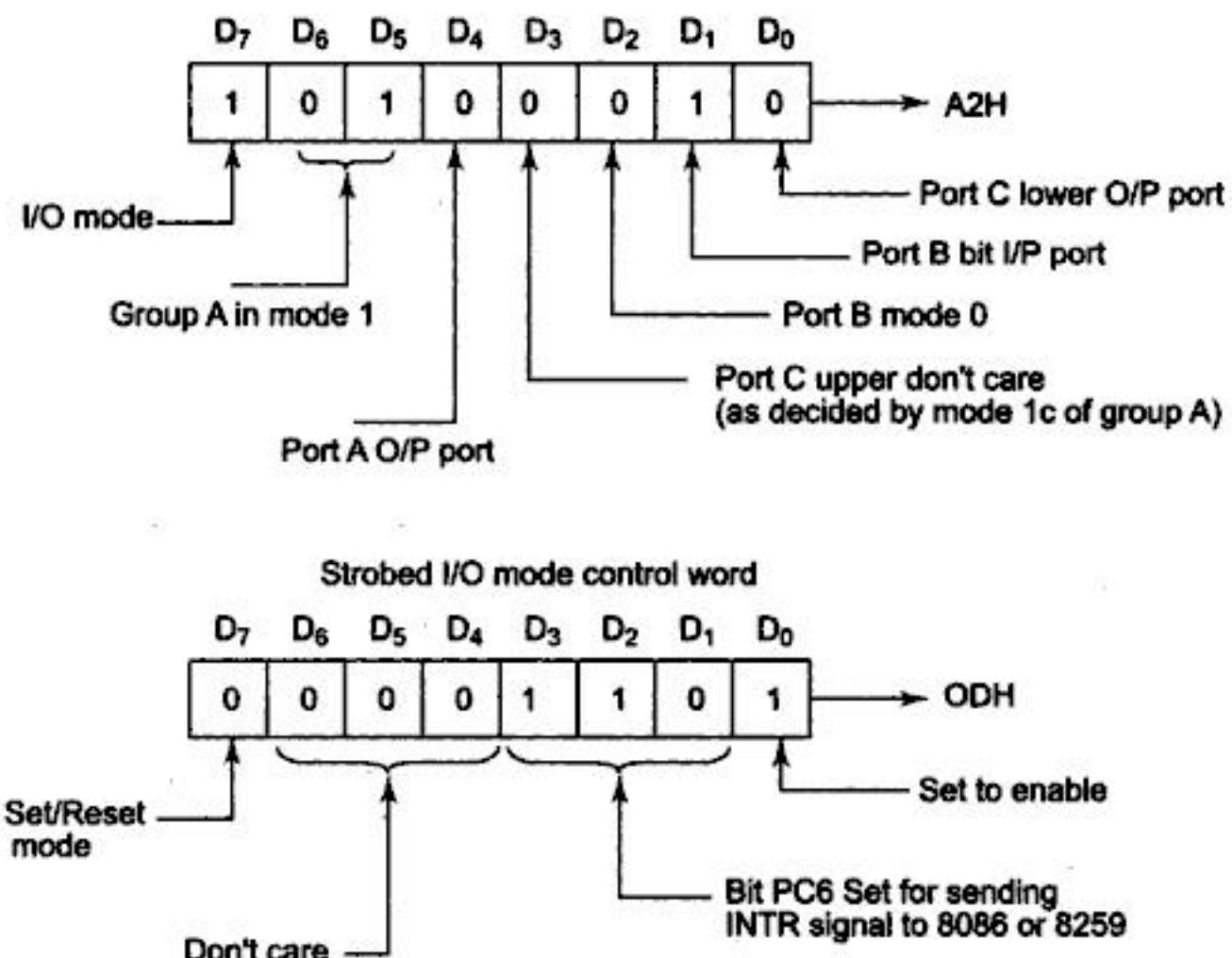


Fig. 5.30 Bit Set/Reset Control Word

asserts its ACKNLG signal low approximately for 5 ms. The rising edge of the ACKNLG signal indicates to the computer that it is ready to receive the next character. The rising edge of the ACKNLG signal also resets the BUSY signal from the printer. A low on the BUSY pin further indicates that the printer is ready to accept the next character. The ACKNLG and BUSY signals can be used interchangeably for handshaking purposes. The waveforms for the above printer operation are shown in Fig. 5.33.

```

MOV BL,AL      ; Get the ASCII code in BL.
MOV AL,0A2H    ; Control word for 8255
OUT OF6H,AL    ; Load CWR with the control word.
BUSY: IN AL,OF2H ; Read printer status from the BUSY pin.
          AND AL,08H ; Mask all bits except PB3
          JNZ BUSY   ; If AL#0, printer is busy. Wait till
                      ; it becomes free.
          MOV AL, BL   ; Get the character for printer in AL
          OUT OF0H,AL  ; Send it to the port for the printer
          NOP          ; Wait for some time
          MOV AL,08 H  ; Pull STROBE low
          OUT OF6H    ; Reset PC4
          NOP          ; Wait
          MOV AL,09 H  ; Raise STROBE high
          OUT OF6H    ; Set PC4
          HLT

```

Program 5.8 ALP for Printing a Character for Problem 5.14

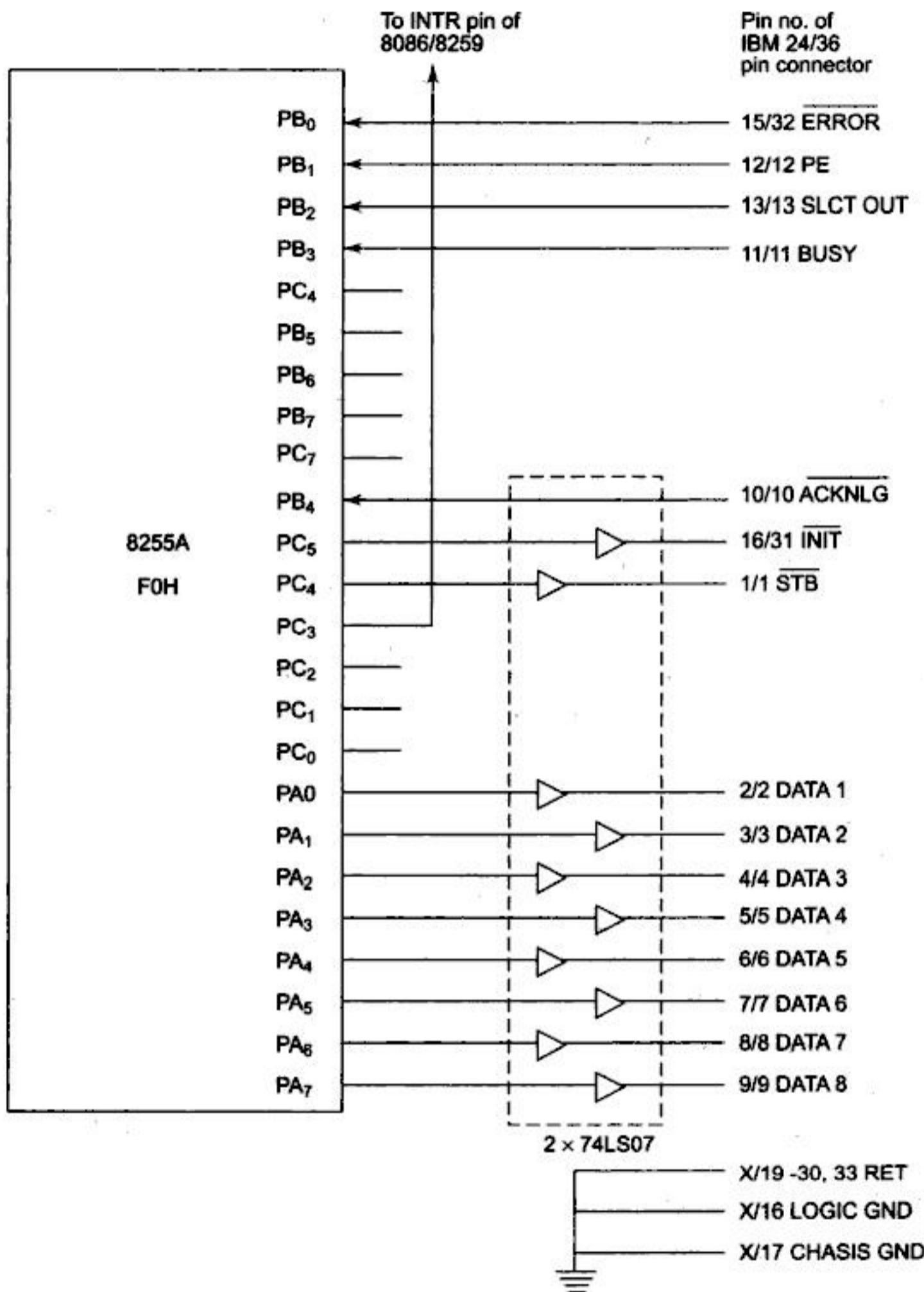
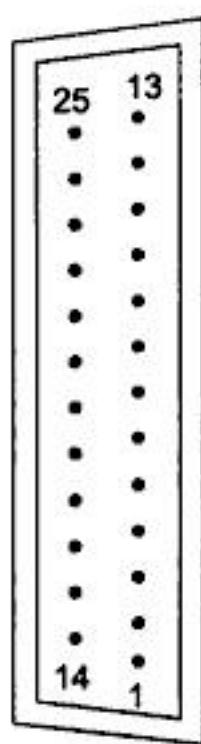
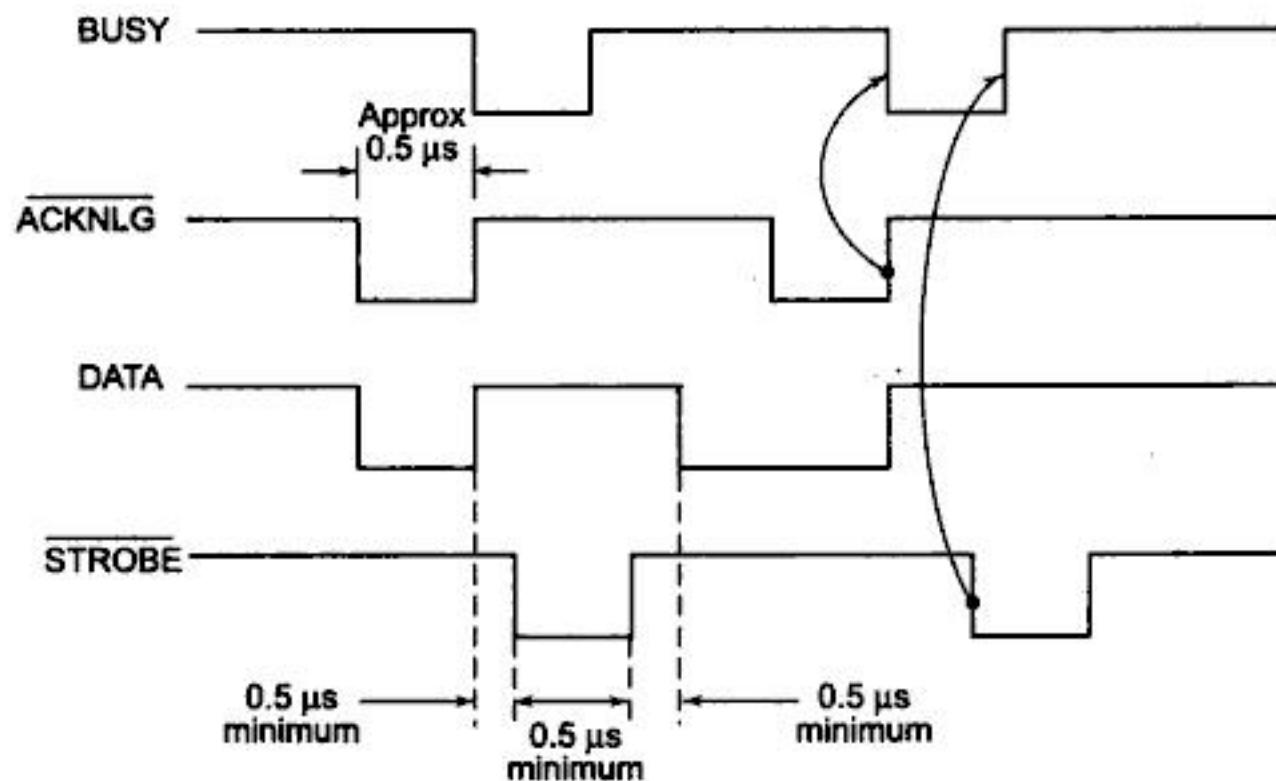


Fig. 5.31 Printer Interface with 8255

**MODE 2 (Strobed bidirectional I/O)** This mode of operation of 8255 is also known as *strobed bidirectional I/O*. This mode of operation provides 8255 with an additional feature for communicating with a peripheral device on an 8-bit data bus. Handshaking signals are provided to maintain proper data



**Fig. 5.32** Centronics Printer Connector



**Fig. 5.33** Timing Waveforms of Data Transfer to a Centronix Compatible Parallel Printer

flow and synchronization between the data transmitter and receiver. The interrupt generation and other functions are similar to mode 1. Thus in this mode, 8255 is a bidirectional 8-bit port with handshake signals. The  $\overline{RD}$  and  $\overline{WR}$  signals decide whether the 8255 is going to operate as an input port or output port.

The salient features of mode 2 of 8255 are listed as follows:

1. The single 8-bit port in group A is available.
2. The 8-bit port is bidirectional and additionally a 5-bit control port is available.
3. Three I/O lines are available at port C, viz.  $PC_2-PC_0$ .
4. Inputs and outputs are both latched.
5. The 5-bit control port C ( $PC_3-PC_7$ ) is used for generating/accepting handshake signals for the 8-bit data transfer on port A.

### **Control signal definitions in mode 2**

**INTR (Interrupt request)** As in mode 1, this control signal is active high and is used to interrupt the microprocessor to ask for transfer of the next data byte to/from it. This signal is used for input (read) as well as output (write) operations.

### **Control signals for output operations**

**OBF (Output buffer full)** This signal, when falls to logic low level, indicates that the CPU has written data to port A.

**ACK (Acknowledge)** This control input, when falls to logic low level, acknowledges that the previous data byte is received by the destination and the next byte may be sent by the processor. This signal enables the internal tristate buffers to send out the next data byte on port A.

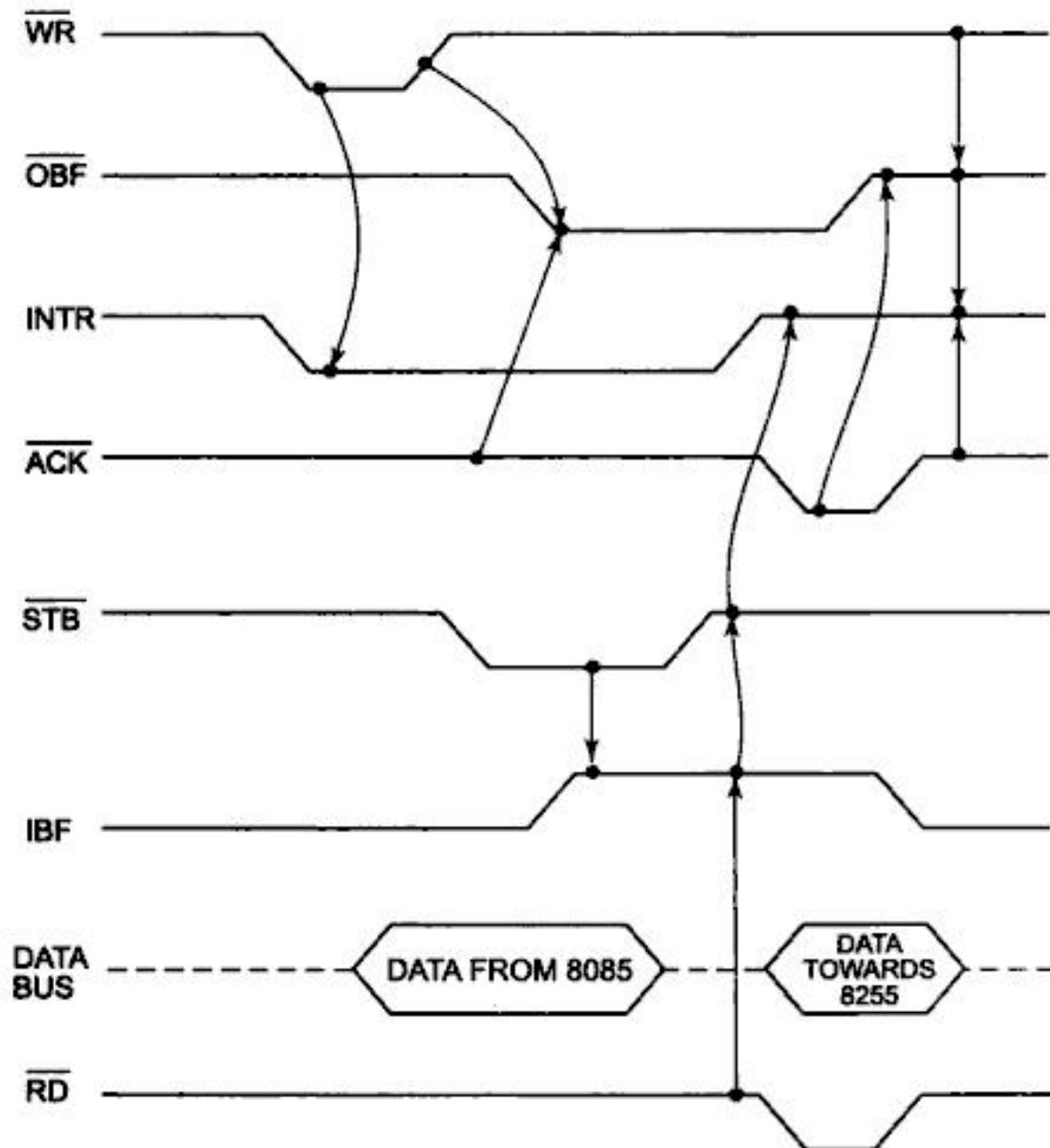
**INTE1 (A flag associated with OBF)** This can be controlled by bit set/reset mode with  $PC_6$ .

### **Control signals for Input operations**

**STB (Strobe input)** A low on this line is used to strobe in the data into the input latches of 8255.

**IBF (Input buffer full)** When the data is loaded into the input buffer, this signal rises to logic '1'. This can be used as an acknowledgement that the data has been received by the receiver.

The waveforms in Fig. 5.34 show the operation in mode 2 for output as well as input port.



**Fig. 5.34 Mode 2 Bidirectional Data Transfer**

**Note:**  $\overline{\text{WR}}$  must occur before  $\overline{\text{ACK}}$  and  $\overline{\text{STB}}$  must be activated before  $\overline{\text{RD}}$ .

Figure 5.35 (a) shows a schematic diagram containing an 8-bit bidirectional port, 5-bit control port and the relation of INTR with the control pins. Port B can either be set to mode 0 or mode 1 while port A (Group A) is in mode 2. Mode 2 is not available for port B. Figure 5.35 (b) shows the necessary control word.

The INTR goes high only if either IBF, INTE2,  $\overline{\text{STB}}$  and  $\overline{\text{RD}}$  go high or  $\overline{\text{OBF}}$ , INTE1,  $\overline{\text{ACK}}$  and  $\overline{\text{WR}}$  go high. The port C can be read to know the status of the peripheral device, in terms of the control signals, using the normal I/O instructions.

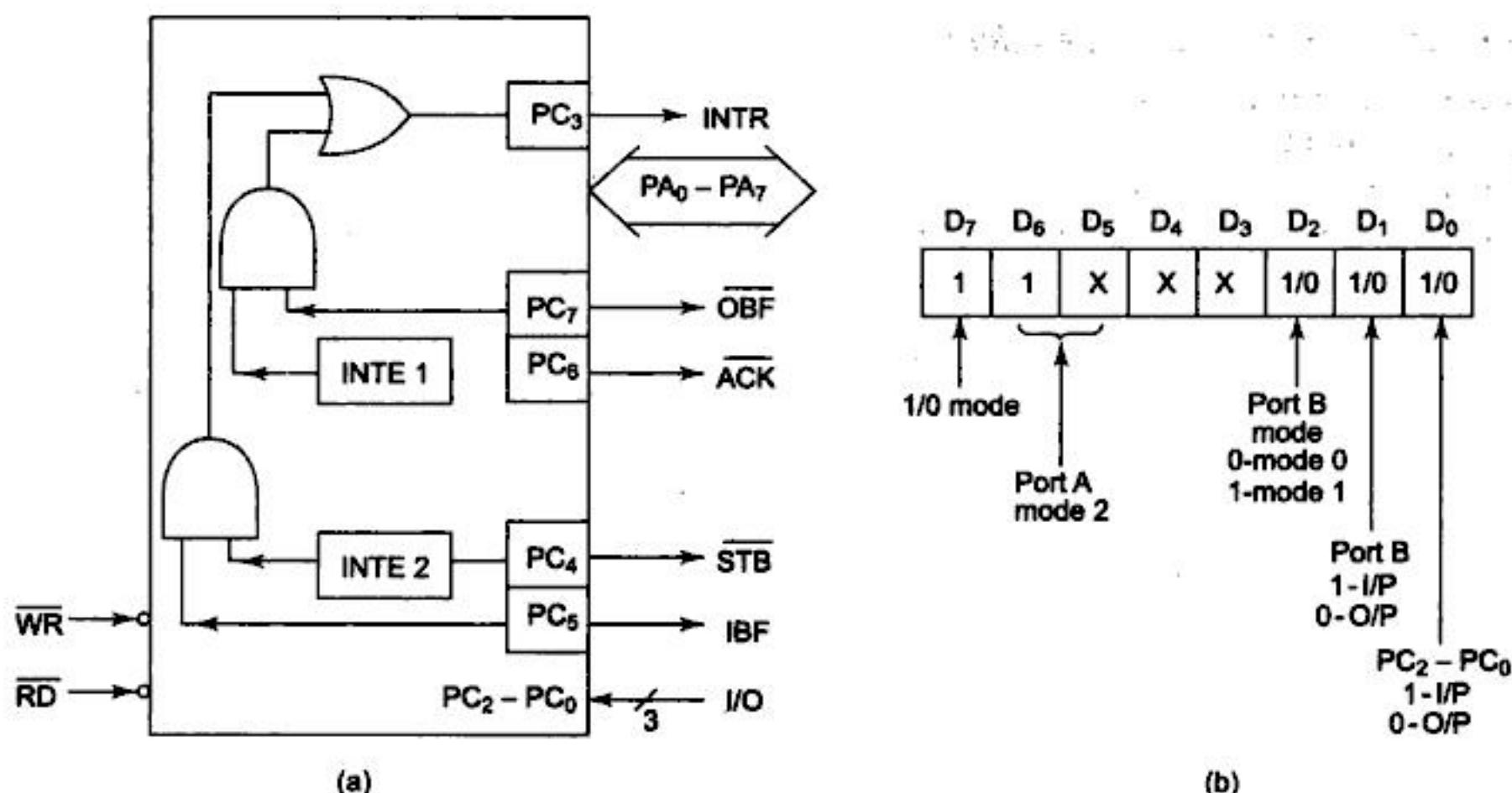


Fig. 5.35 (a) Mode 2 Pins (b) Mode 2 Control Word

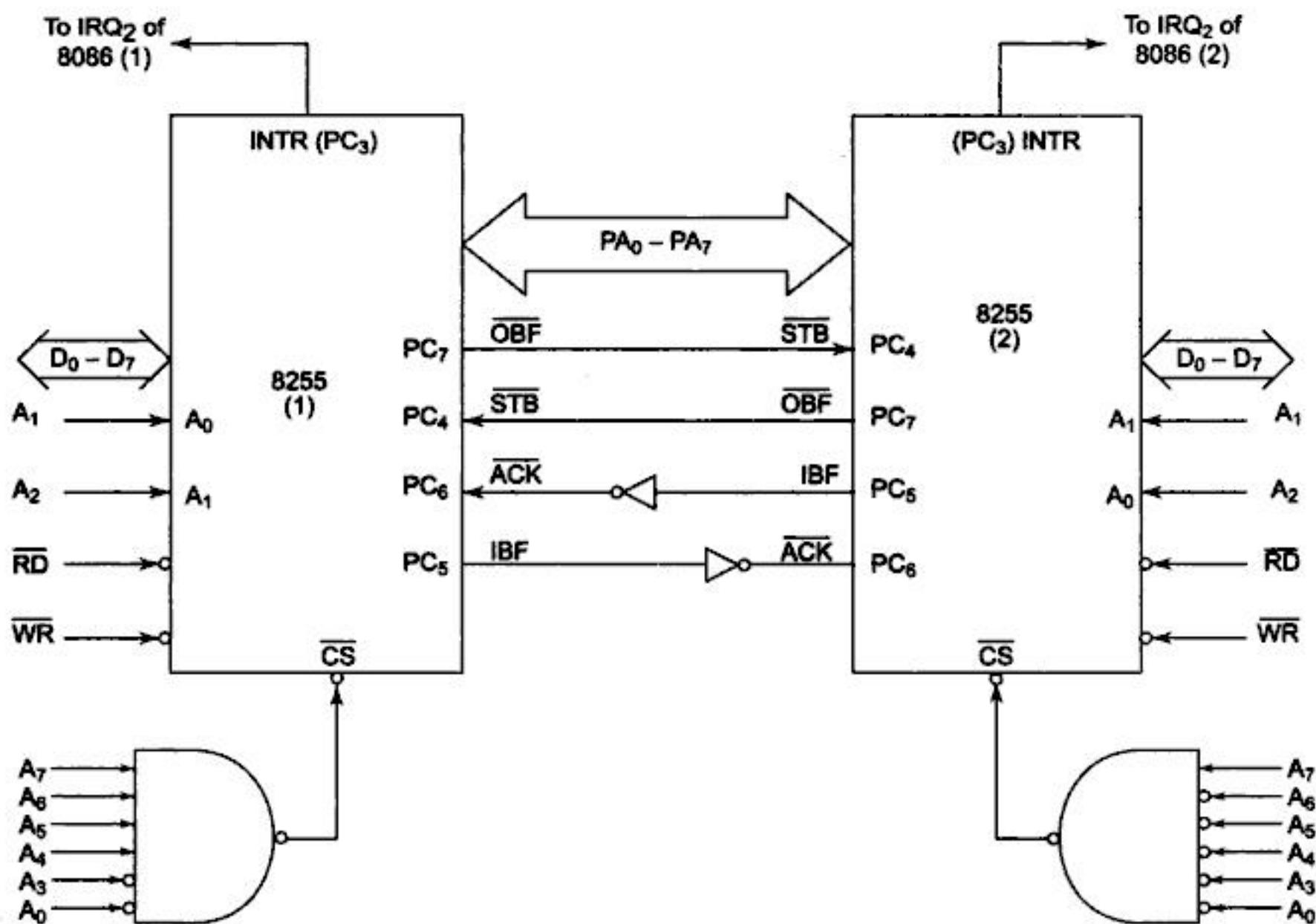
The following problem emphasizes the use of 8255 in mode 2.

### Problem 5.15

An 8086 system with a 8255 interfaced at port A address F0H, has a block of 100 data bytes stored in it. Another 8086 system with another 8255 interfaced at port A address 80H has another block of 100 data bytes stored in it. Interchange these blocks of data bytes between the two 8086 systems. Draw the necessary hardware scheme and write the necessary sequence of instructions. Both systems run on the same CLK rate.

**Solution** The complete hardware schematic is shown in Fig. 5.36. The INTR pin of 8255 in mode 2 is applied to the respective 8086 processor at NMI pin. The inverted IBF signal of the first 8255 is connected to ACK of the second and the inverted IBF signal of the second 8255 is connected to ACK of the first 8255, so that input buffer full signal of one 8255 acknowledges the receipt of data byte sent by the other. The OBF signal of one 8255 is connected to STB signal of the other 8255 and vice versa, so that the output buffer full signal of an 8255 informs the other 8255 that the data is ready on the data bus for it. The 8-bit data bus, i.e. PA<sub>0</sub>-PA<sub>7</sub>, of the two 8255s are connected with each other.

The programming in assembly language is divided into two parts. The first is the transmitter or sender part and the second being the receiver part. The transmitter program sends data to the receiver 8086 system while the receiver program accepts the data from the transmitting 8086 system. To interchange the blocks of data between the two systems, each one will require the transmitter as well as the receiver program.



**Fig. 5.36** Interconnections between the two 8255s in Mode 2 for Problem 5.16

The interrupt vector address for NMI is 0000 : 0008H. At this location the IP and CS values of the transmitter program addresses are stored. The execution of the transmitter program on one system causes execution of the receiver program on the other system through NMI interrupt. Rather, the receiver is executed as the NMI interrupt service routine as a response to the execution of the transmitter program.

; This program transmits parallel data byte by byte to another system through  
; 8255 in mode 2.

```

DATA SEGMENT
CW1 EQU COH
BLOCK1 DB 100D DUP (?)
DATA ENDS
ASSUME CS : CODE, DS : DATA
CODE SEGMENT
MOV AX, 0000H ; Initialise interrupt
MOV DS, AX ; vector table of
MOV AX, OFFSET TRANS ; 8086 at table
MOV [0008H], AX ; TRANS.
MOV [000AH], SEG TRANS
MOV CL, 101D ; count CL (one additional)

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

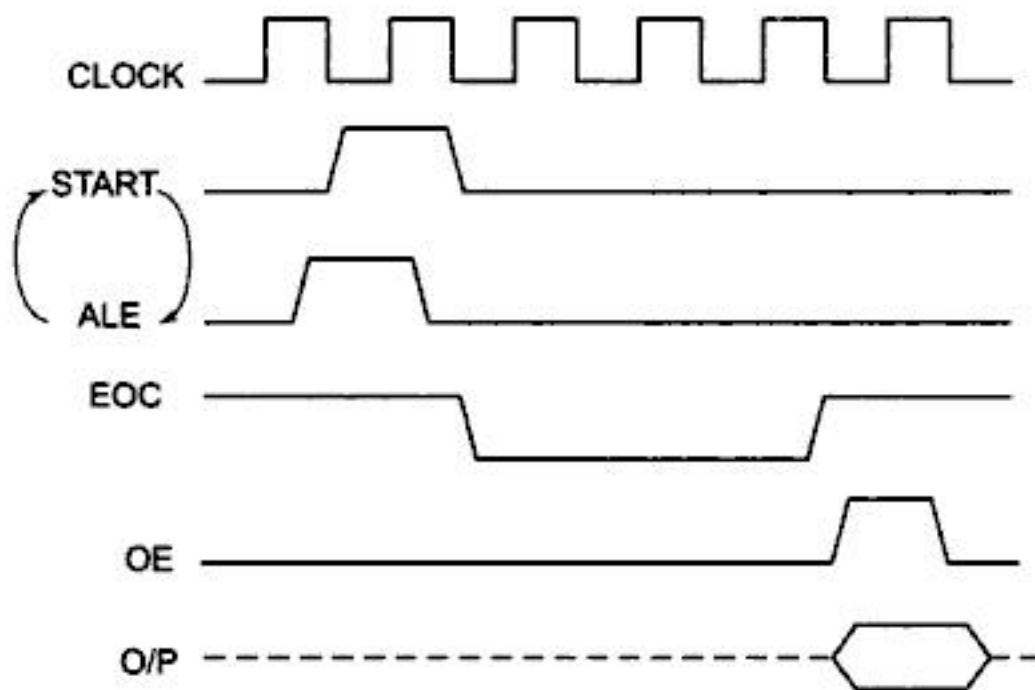


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The timing diagram of different signals of ADC0808 is shown in Fig. 5.38.



**Fig. 5.38 Timing Diagram of ADC 0808**

**Solution** Figure 5.39 shows the interfacing connections of ADC0808 with 8086 using 8255. The analog input I/P<sub>2</sub> is used and therefore address pins A,B,C should be 0,1,0 respectively to select I/P<sub>2</sub>. The OE and ALE pins are already kept at +5V to select the ADC and enable the outputs. Port C upper acts as the input port to receive the EOC signal while port C lower acts as the output port to send SOC to the ADC. Port A acts as a 8-bit input data port to receive the digital data output from the ADC. The 8255 control word is written as follows:

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	Control word
1	0	0	1	1	0	0	0	= 98 H

The required ALP is given as follows:

```

MOV AL,98 H      ; Initialise 8255 as
OUT CWR,AL       ; discussed above
MOV AL,02H       ; Select I/P2 as analog
OUT Port B,AL    ; input
MOV AL,00H       ; Give start of conversion
OUT Port C,AL    ; pulse to the ADC.
MOV AL,01 H      ;
OUT Port C,AL    ;
MOV AL,00H       ;
OUT Port C,AL    ;
WAIT : IN AL,PortC ; Check for EOC by
RCL             ; reading port C upper and
JNC WAIT        ; rotating through carry.
IN AL,PortA      ; If EOC, read digital equivalent in AL
HLT             ; Stop

```

**Program 5.11 ALP for Problem 5.16**

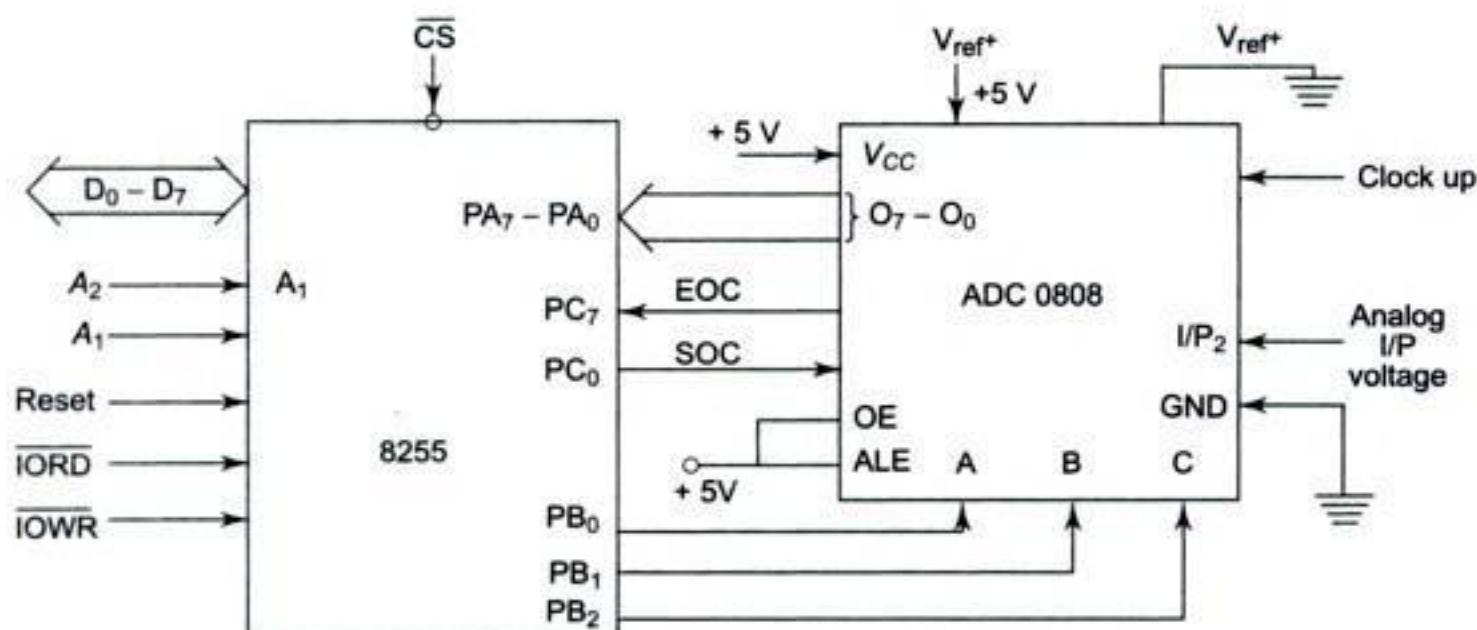


Fig. 5.39 Interfacing 0808 with 8086

### 5.6.2 ADC 7109—A Dual Slope 12-bit ADC

Intersil's ICL 7109 is a dual slope integrating analog to digital converter. This 12-bit ADC is designed to work with 8-bit and 16-bit or higher order microprocessors with great ease. As compared to other 12-bit ADCs, it is a very low cost option, useful for slow practical applications, as the method used for analog to digital conversion is dual slope integration.

The 12-bit data output, polarity and overrange signals can be directly accessed under the software control of two byte enable inputs LBEN and HBEN. The RUN/HOLD and STATUS outputs allow monitoring and control of the conversion process. The salient features of the ADC 7109 are as given as follows:

1. 12-bit data output equivalent to analog input along with polarity, over range and under range outputs
2. It can be operated in parallel or serial output mode
3. Differential input and differential reference
4. Low noise (Typical 15  $\mu$ V p-p)
5. Low input current (Typical 1pA)
6. Can operate up to 30 conversions per second, with an external crystal or RC circuit used to decide the operating clock frequency.

The pin diagram of the ADC is given here followed by brief signal descriptions, in Fig. 5.40 and Table 5.14 respectively.

Table 5.14 Pin Assignment and Function Description

Pin	Symbol	Description
1.	GND	Digital Ground return for all digital logic
2.	STATUS	Output High during integrate and deintegrate until data is latched Output Low when analog section is in Auto-Zero configuration
3.	POL	Polarity-HI for Positive input
4.	OR	Over range-HI if Overranged

(Contd.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

<i>Pin</i>	<i>Symbol</i>	<i>Description</i>
		Input High—Enables <u>CE/LOAD</u> (Pin 20), <u>HBEN</u> (Pin 19), and <u>LBEN</u> (Pin 18) as outputs, handshake mode will be entered and data output is available after conversion completion.
22.	OSC IN	Oscillator Input
23.	OSC OUT	Oscillator Output
24.	OSC SEL	Oscillatory Select—Input high configures OSC IN, OSC OUT, BUF OSC OUT as RC oscillator—clock will be of same phase and duty cycle as BUF OSC OUT. —Input low configures OSC IN, OSC OUT for crystal oscillator—clock frequency will be 1/58 of frequency at BUF OSC OUT.
25.	BUF OSC OUT	Buffered Oscillator Output
26.	RUN/HOLD	Input High—Conversion continuously performed every 8 192 clocks pulses. Input Low—converter will stop in Auto-Zero 7 counts before integrate.
27.	SEND	Input—Used in handshake mode to indicate ability of an external device to accept data. Connect to + 5V if not used.
28.	V <sup>-</sup>	Analog Negative Supply—Nominally -5V with respect to GND (Pin 1).
29.	REF OUT	Reference Voltage Output—Nominally 2.88 V down from V <sup>-</sup> (Pin 40).
30.	BUFFER	Buffer Amplifier Output
31.	AUTO-ZERO	Auto-Zero Node—Inside foil of CAZ
32.	INTEGRATOR	Integrator Output—Outside foil of C <sub>INT</sub>
33.	COMMON	Analog Common—System is Auto-Zeroed to COMMON
34.	INPUT LO	Differential Input Low Side
35.	INPUT HI	Differential Input High Side
36.	REF IN +	Differential Reference Input Positive
37.	REF CAP +	Reference Capacitor Positive
38.	REF CAP -	Reference Capacitor Negative
39.	REF IN	Differential Reference Input Negative
40.	V <sup>+</sup>	Positive Supply Voltage—Nominally + 5V with respect to GND (Pin 1).

**Note:** All digital levels are positive true.

GND	1		40	V+
STATUS	2		39	REF IN-
POLARITY	3		38	REF CAP-
OVER RANGE	4		37	REF CAP+
B <sub>12</sub>	5		36	REF IN+
B <sub>11</sub>	6		35	IN HI
B <sub>10</sub>	7		34	IN LO
B <sub>9</sub>	8		33	COMMON
B <sub>8</sub>	9		32	INT
B <sub>7</sub>	10	ICL7109	31	AZ
B <sub>6</sub>	11		30	BUF
B <sub>5</sub>	12		29	REF OUT
B <sub>4</sub>	13		28	V-
B <sub>3</sub>	14		27	SEND
B <sub>2</sub>	15		26	RUN/HOLD
B <sub>1</sub>	16		25	BUF OSC OUT
TEST	17		24	OSC SELECT
LBEN	18		23	OSC OUT
HBEN	19		22	OSC IN
CE/LOAD	20		21	MODE

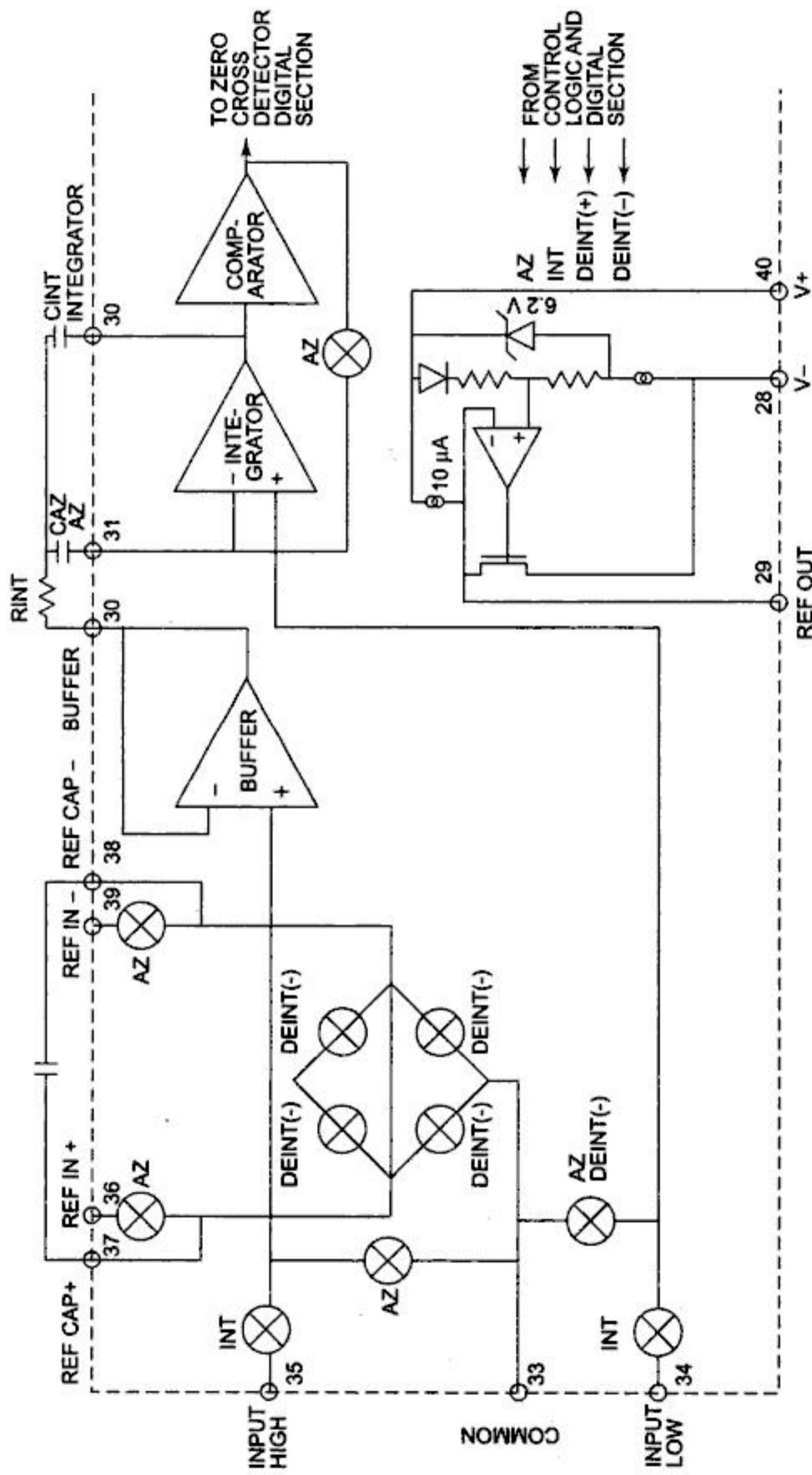
**Fig. 5.40** Pin Configuration of ICL 7109ADC

The internal circuit of ICL 7109 is slightly complicated. Hence it is divided in two parts namely—analog section and digital section for better understanding. The following text explains the internal operation in terms of the separate sections.

**Analog Section** The block diagram of the internal analog section of ICL 7109 is shown in Fig. 5.41. If RUN/HOLD is either left open or connected to  $+V_{cc}$ , the ADC starts conversion at the rate determined by the clock frequency, to be decided either by a crystal or by an RC circuit. The total conversion cycle is divided into three phases namely *autozero phase*, *signal integrate phase* and *deintegrate phase*.

**Autozero phase** During autozero phase, input high (IN HI) and input low (IN LO) are disconnected from the external input signal and shorted to analog common. Then the reference capacitor is charged to reference voltage. A feedback loop is closed around the system to charge the autozero capacitor CAZ to compensate for the offset voltages in the buffer amplifier, integrator and comparator.

**Signal integrate phase** In this phase the autozero capacitor CAZ is opened out. The external input signal is connected with the internal circuit. The differential input voltage between IN LO and IN HI pins is then integrated by the internal integrator for a fixed period of 2048 clock cycles.



**Fig. 5.41** Analog Section of 7109



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

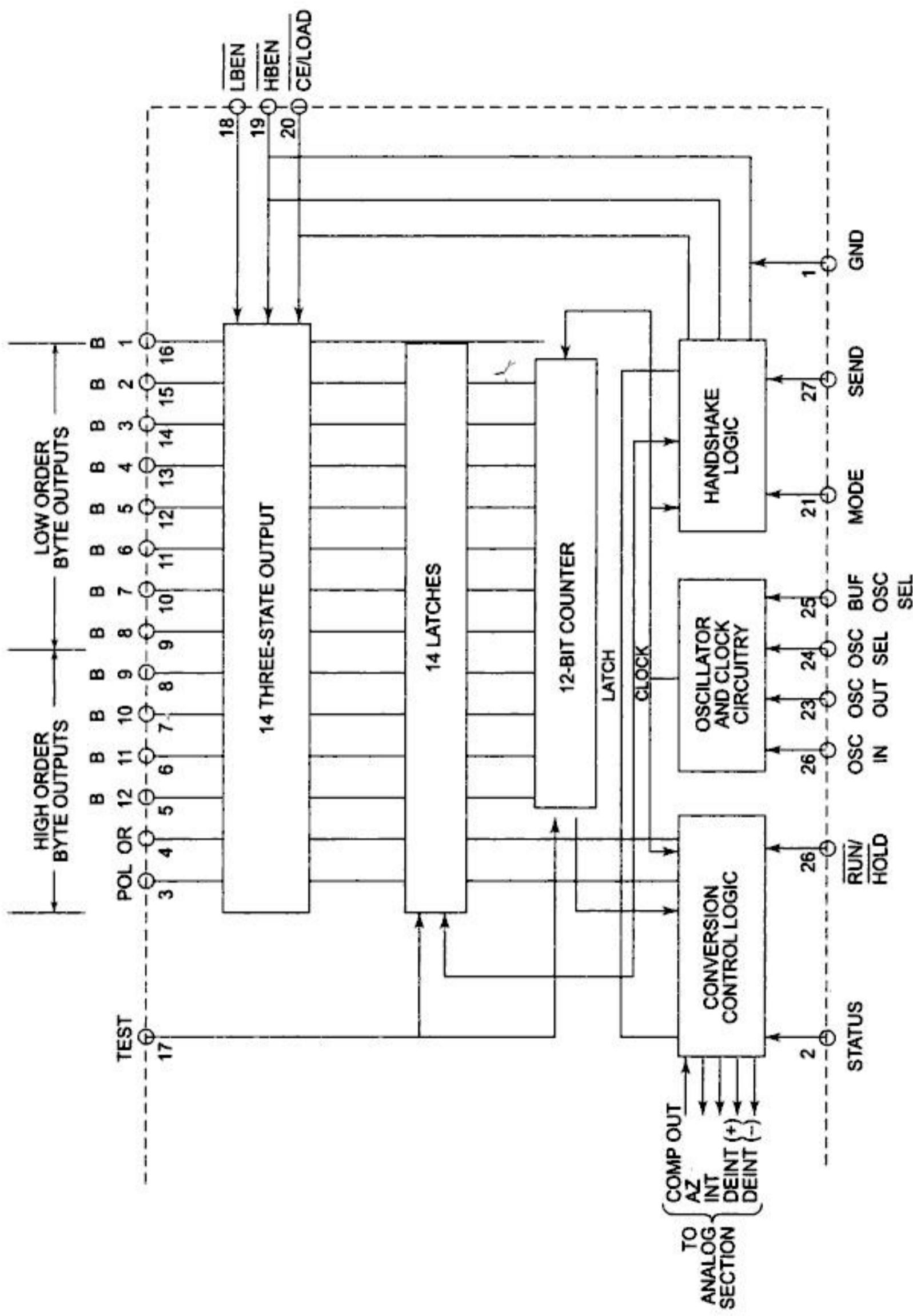
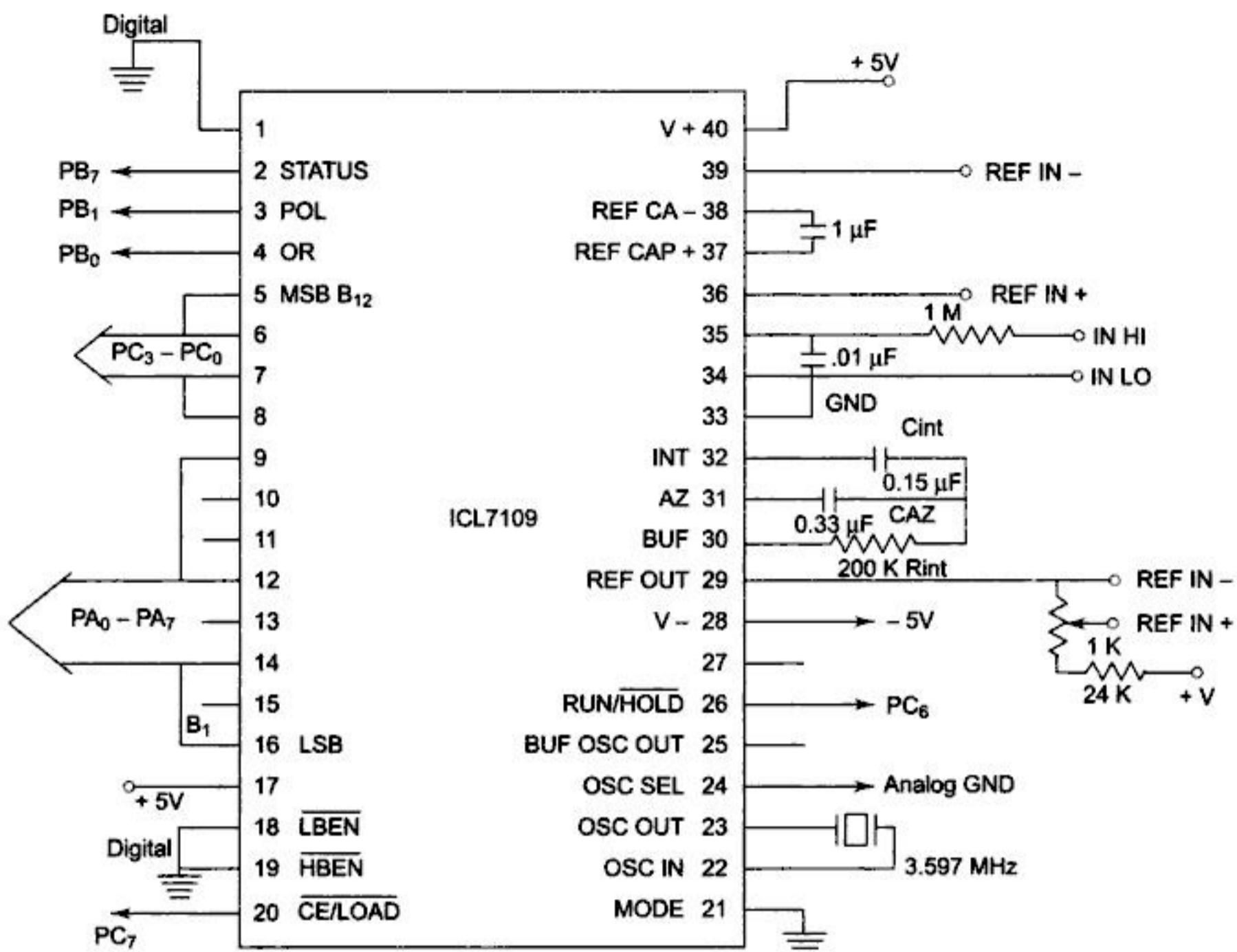


Fig. 5.42 Digital Section of ICL 7109



**Fig. 5.43** Interfacing 7109 with 8086 through 8255

The parallel I/O port chip 8255 is used to interface ICL 7109 with 8086. Being a 12-bit ADC, it will require 12 I/O lines for data outputs. Port A and Port C lower are used as input ports to read digital data output from the ADC. The pins LBEN and HBEN are permanently grounded to enable all the data lines from  $B_1$  to  $B_{12}$  at a time. The lines CE/LOAD and RUN/HOLD are controlled using port C upper of 8255, that is used as an output port and finally port B is used to read the STATUS, POL and OR signals and hence is used as an input port. The polarity (POL) and overrange data (OR) will be available in DL register with  $D_0$  corresponding to OR and  $D_1$  corresponding to POL. The digital data is read from the ports if STATUS goes low, i.e. conversion if the over. Figure 5.44 shows the algorithm for the analog to digital conversion with ADC 7109. After the execution of the program the digital equivalent of the analog input is in register CX, while the overrange and polarity information is in DL. From the above description 8255 control word comes out to be 93H.

The above circuit components are designed for 4096 mV full scale analog input range. As specified in the manual the Rint for this range is 200 K. The reference input voltage between REF IN- and REF IN+ will be half of full scale analog input voltage.

```

; This program issues a RUN signal to ICL7109, checks for
; STATUS(EOC) and reads digital output with POL and OR outputs.

ASSUME CS : CODE
CODE SEGMENT
START: MOV AL, 93H      ; Initialization of
        OUT CWR, AL    ; 8255
        MOV AL, 40 H     ; RUN (PC6) to go high and
        OUT Port C, AL   ; CE(PC7) to go low, for start of conversion.
        WAIT : IN AL, PortB ; Read STATUS signal.
        RCL AL, 01       ; Check STATUS using carry flag.
        JC WAIT          ; Wait till carry (STATUS) goes
        IN AL, PortA     ; low i.e. conversion is over.
        MOV CL, AL
        IN AL, PortC     ; Read digital data output and store the
        AND AL, 0FH       ; lower byte in CL and higher byte in CH. Mask
        MOV CH, AL         ; higher bits of CH as of only 12 bits are of interest.
        IN AL, PortB     ; Store OR and POL in D0 and D1 in
        MOV DL, AL         ; DL register.
        MOV AH, 4CH       ; Return to DOS.
        INT 21H
CODE ENDS
END START

```

#### **Program 5.12 ALP for Interfacing ICL 7109 with 8086**

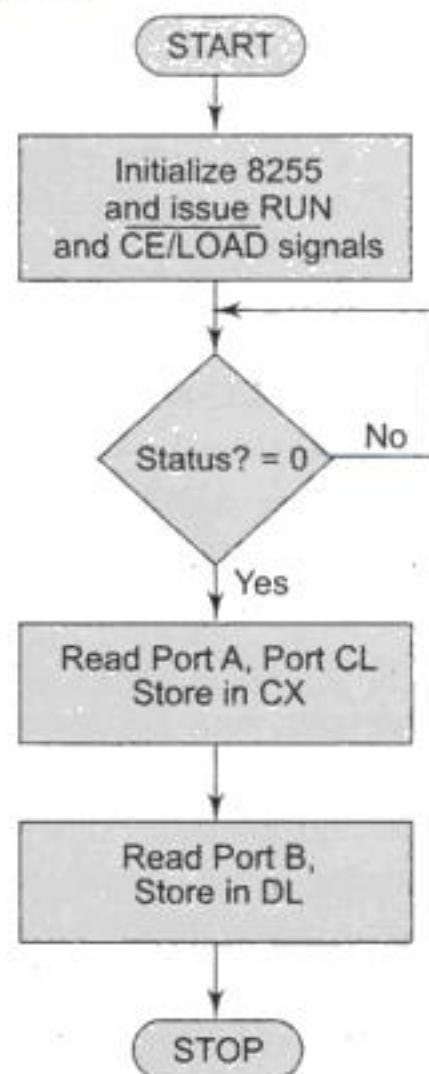
Readers may find a number of other 8-bit and 12-bit ADCs, their details and interfacing techniques from the respective data manuals or other textbooks. The discussion here is mainly aimed at explaining the general interfacing techniques of ADCs though the specific ADCs are discussed in detail.

### **5.7 INTERFACING DIGITAL TO ANALOG CONVERTERS**

The digital to analog converters convert binary numbers into their analog equivalent voltages. The DAC find applications in areas like digitally controlled gains, motor speed controls, programmable gain amplifiers, etc. This text explains the generally available DAC integrated circuits and their interfacing techniques with the microprocessor.

#### **5.7.1 AD 7523 8-bit Multiplying DAC**

Intersil's AD 7523 is a 16 pin DIP, multiplying digital to analog converter, containing R-2R ladder ( $R = 10\text{ K}$ ) for digital to analog conversion along with single pole double throw NMOS switches to connect the digital inputs to the ladder.



**Fig. 5.44 Algorithm for reading O/P of 7109**



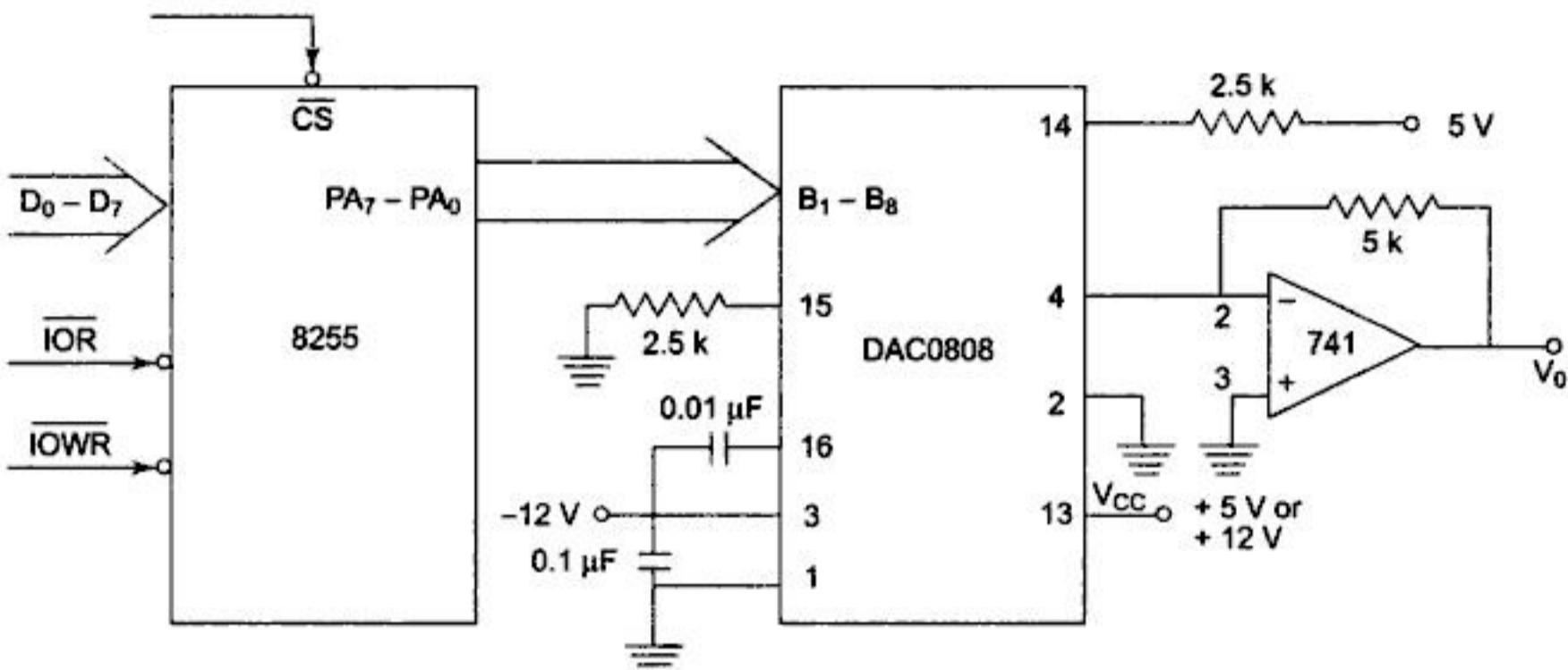
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Problem 5.18**

Write an assembly language program to generate a triangular wave of frequency 500 Hz using the interfacing circuit given in Fig. 5.48. The 8086 system operates at 8 MHz. The amplitude of the triangular wave should be +5 V.



**Fig. 5.48** Interfacing DAC0800 with 8086

**Solution** The  $V_{ref+}$  should be tied to +5 V to generate a wave of +5V amplitude. The required frequency of the output is 500 Hz, i.e. the period is 2 ms. Assuming the wave to be generated is symmetric, the waveform will rise for 1 ms and fall for 1 ms. This will be repeated continuously. In the previous program, we have already written an instruction sequence for period 1 ms. Using the same instruction sequence one can derive this triangular waveform. The ALP is given as follows:

```

ASSUME    CS : CODE
CODE      SEGMENT
START     MOV AL,80 H      ; Initialise 8255 ports
          OUT CWR,AL      ; suitably.
          MOV AL,00H      ; Start rising ramp from
BACK      OUT Port A,AL   ; 0V by sending 00H to DAC.
          INC AL        ; Increment ramp till 5V
          CMP AL,FFH   ; i.e. FFH.
          JB BACK      ; If it is FFH then,
BACK1     OUT Port A,AL   ; Output it and start the falling
          DEC AL        ; ramp by decrementing the
          CMP AL,00    ; counter till it reaches
          JA BACK1    ; zero. Then start again
          JMP BACK    ; for the next cycle.
CODE      ENDS
END START

```

**Program 5.14** ALP for Generating a Triangular Wave Using DAC 0800

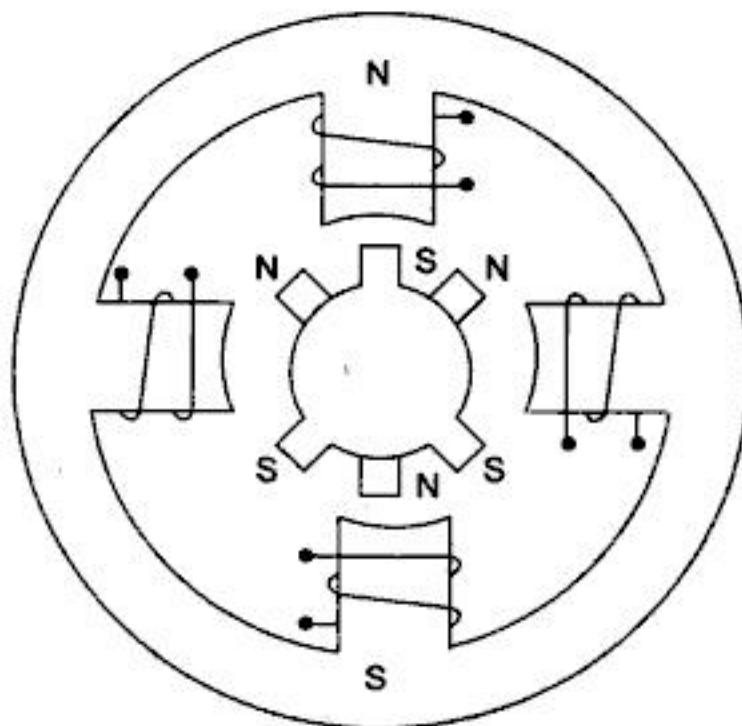
The technique of interfacing 12-bit DACs with 8086 is also similar. If 8-bit ports are used for interfacing a 12-bit DAC, two successive 8-bit OUT instructions are required to apply input to the DAC. If a 16-bit port is used for interfacing a 12-bit DAC, a single OUT instruction is sufficient to apply the 12-bit input to the DAC. 12-bit DACs generate more precise analog voltages ( $2^{12} = 4096$  steps in full output range) as compared to 8-bit DACs ( $2^8 = 256$  steps in full output range).

## 5.8 STEPPER MOTOR INTERFACING

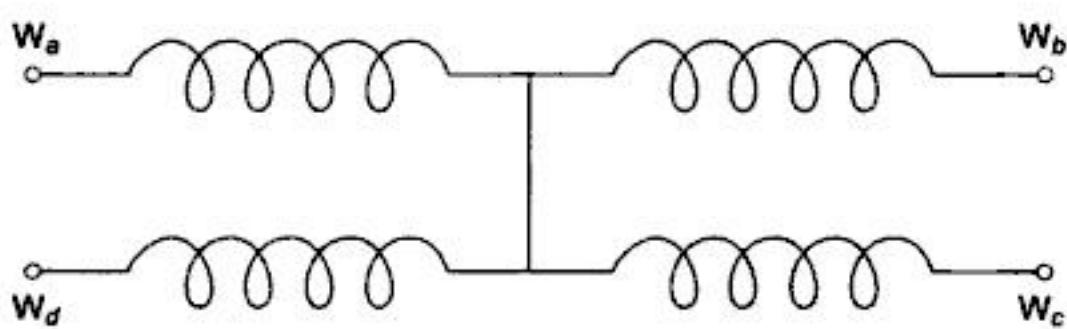
A stepper motor is a device used to obtain an accurate position control of rotating shafts. It employs rotation of its shaft in terms of steps, rather than continuous rotation as in case of AC or DC motors. To rotate the shaft of the stepper motor, a sequence of pulses is needed to be applied to the windings of the stepper motor, in a proper sequence. The number of pulses required for one complete rotation of the shaft of the stepper motor are equal to its number of internal teeth on its rotor. The stator teeth and the rotor teeth lock with each other to fix a position of the shaft. With a pulse applied to the winding input, the rotor rotates by one teeth position or an angle  $x$ . The angle  $x$  may be calculated as:

$$x = 360^\circ / \text{no. of rotor teeth}$$

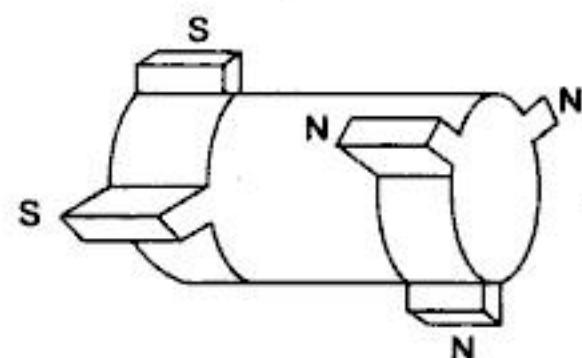
After the rotation of the shaft through angle  $x$ , the rotor locks itself with the next tooth in the sequence on the internal surface of stator. The internal schematic of a typical stepper motor with four windings is shown in Fig. 5.49(a). The stepper motors have been designed to work with digital circuits. Binary level pulses of 0-5V are required at its winding inputs to obtain the rotation of shafts. The sequence of the pulses can be decided, depending upon the required motion of the shaft. Figure 5.49(b) shows a typical winding arrangement of the stepper motor. Figure 5.49(c) shows conceptual positioning of the rotor teeth on the surface of rotor, for a six teeth rotor.



**Fig. 5.49(a)** Internal Schematic of a Four Winding Stepper Motor



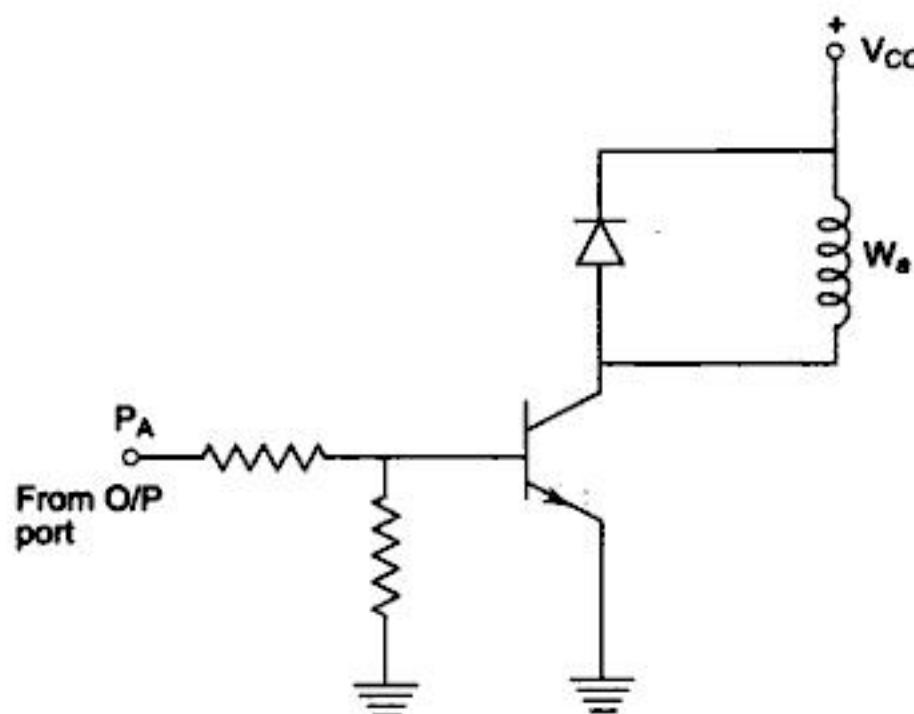
**Fig. 5.49(b)** Winding Arrangement of a Stepper Motor



**Fig. 5.49(c)** A Stepper Motor Rotor

The circuit for interfacing a winding  $W_n$  with an I/O port is given in Fig. 5.50. Each of the windings of a stepper motor need this circuit for its interfacing with the output port. A typical stepper motor may

have parameters like torque 3 kg-cm, operating voltage 12 V, current rating 0.2 A and a step angle 1.8°, i.e. 200 steps/revolution (number of rotor teeth).



**Fig. 5.50** Interfacing Stepper Motor Winding  $W_a$

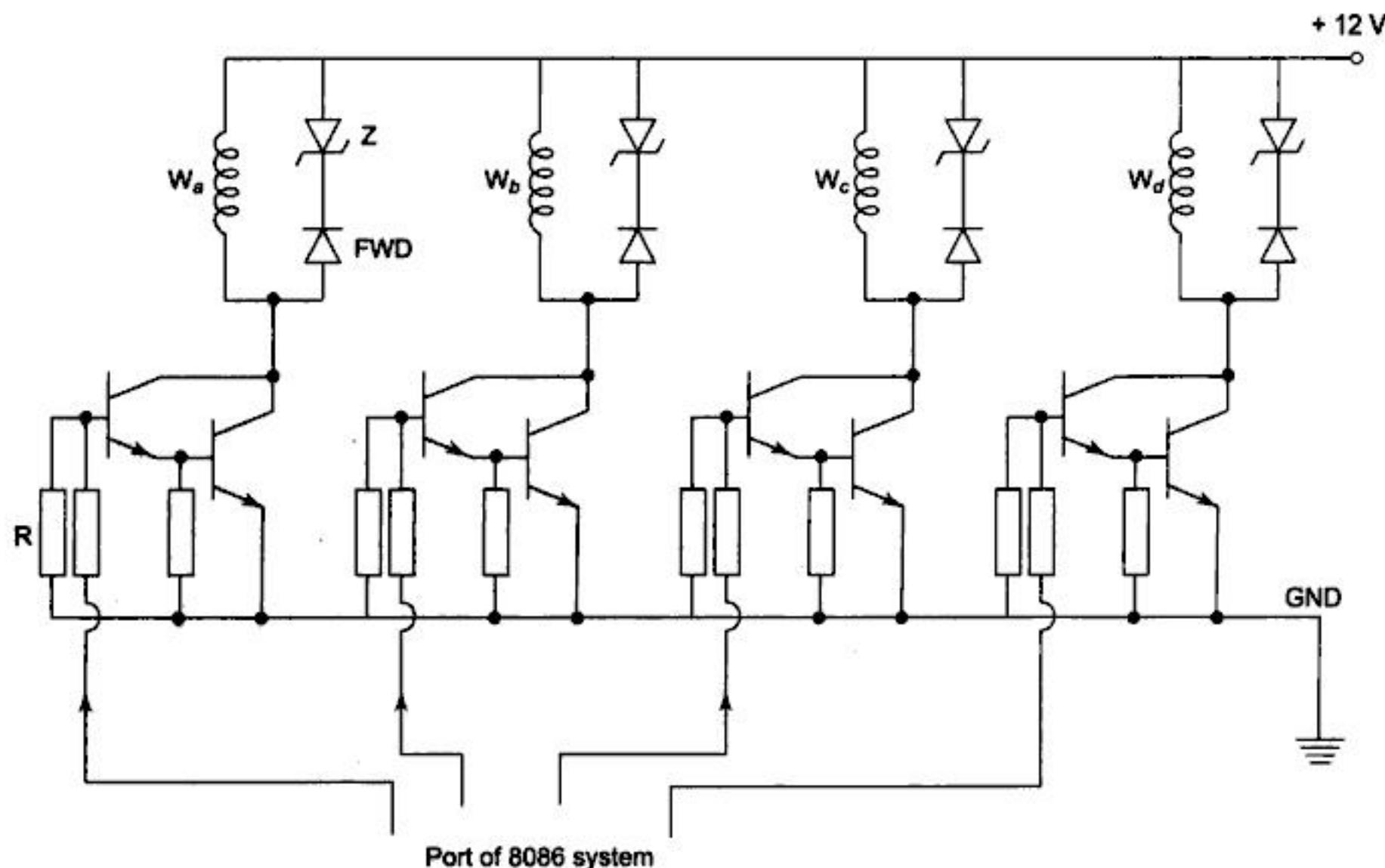
A simple scheme for rotating the shaft of a stepper motor is called a wave scheme. In this scheme, the windings  $W_a$ ,  $W_b$ ,  $W_c$  and  $W_d$  are applied with the required voltage pulses, in a cyclic fashion. By reversing the sequence of excitation, the direction of rotation of the stepper motor shaft may be reversed. Table 5.15(a) shows the excitation sequences for clockwise and anticlockwise rotations. Another popular scheme for rotation of a stepper motor shaft applies pulses to two successive windings at a time but these are shifted only by one position at a time. This scheme for rotation of stepper motor shaft is shown in Table 5.15(b).

**Table 5.15(a)** Excitation Sequences of a Stepper Motor Using Wave Switching Scheme

Motion	Step	A	B	C	D
Clockwise	1	1	0	0	0
	2	0	1	0	0
	3	0	0	1	0
	4	0	0	0	1
	5	1	0	0	0
Anticlockwise	1	1	0	0	0
	2	0	0	0	1
	3	0	0	1	0
	4	0	1	0	0
	5	1	0	0	0



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Fig. 5.51 Stepper Motor Windings Connections**

```

AGAIN2:    OUT PORTA,AL      ; Excite WA, WB, WC, and WD.
CALL DELAY
ROR AL, 01      ; Wait.
DEC CX          ; Rotate bit pattern right to obtain anticlockwise
JNZ AGAIN2      ; motion of shaft. Decrement count.
MOV AH, 4CH      ; If 5 rotations are completed
INT 21H          ; return to DOS else
CODE           ; Continue
ENDS
END START

```

**Program 5.15 ALP for Problem 5.19**

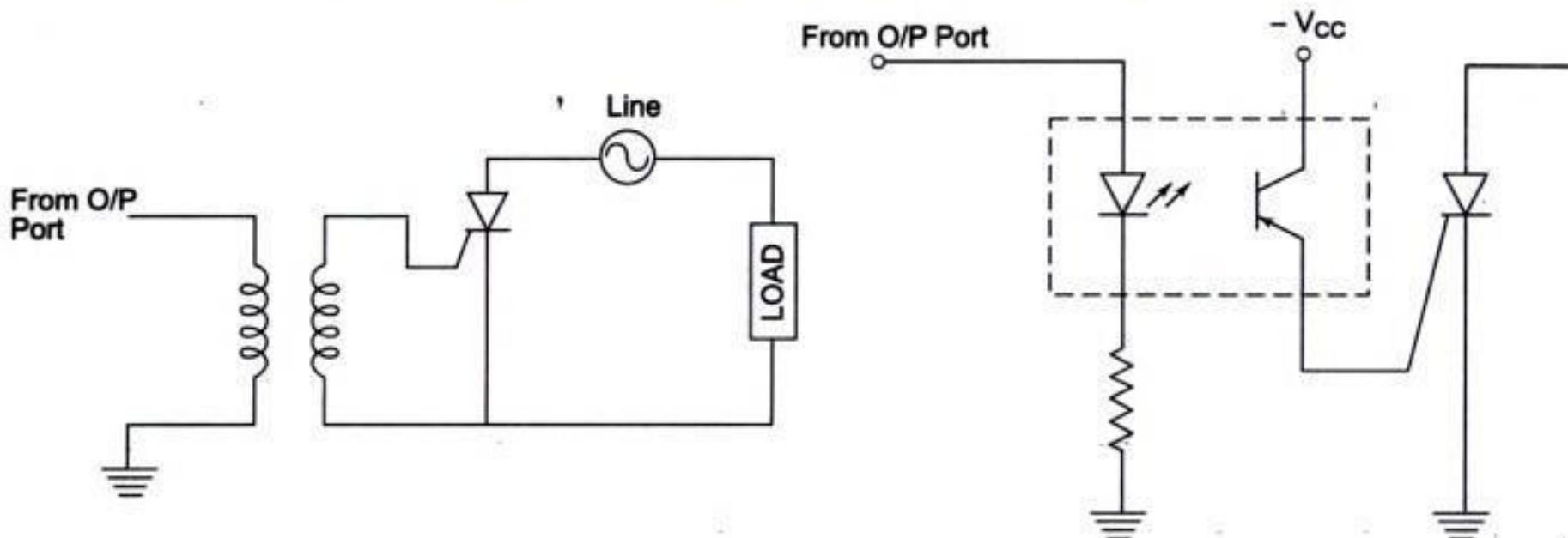
The count for rotating the shaft of the stepper motor through a specified angle may be calculated from the number of rotor teeth. The number of rotor teeth is equal to the count for one rotation, i.e. 360°. Hence for any specified angle  $\theta^\circ$  the count is calculated as:

$$C = \frac{\text{Number of rotor teeth}}{360^\circ} \times \theta^\circ$$

**5.8 CONTROL OF HIGH POWER DEVICES USING 8255**

High power devices like motors, heating furnaces, temperature baths and other process equipments may be controlled using I/O ports of a microcomputer system. These are controlled using power

electronics devices like SCRs, triacs, etc. These devices control the flow of energy to the processes to be controlled. This is obtained by controlling the firing angle of SCRs or Triacs. In the early days, the firing angles of thyristors were controlled using pulse generating circuits like relaxation oscillators but with the advancement in the field of microprocessors, it was observed that SCRs or triacs can be more accurately fired using a microprocessor. The main problem in this method lies in isolation between low power and high power circuits. A microprocessor circuit is a low power circuit, while an electrical circuit of a furnace is a high power circuit. Even any switching component of a high power circuit may be sufficient to damage the microprocessor system. Hence to protect this low power circuit, isolation is necessary. Isolation transformers are generally used for this purpose. Optoisolators also provide excellent isolation between high power and low power sides. Moreover, they are compact in size and cost effective as compared to the pulse transformers. The I/O signals generated by the microprocessor for these high power devices are applied through these isolating circuits as shown in Figs 5.52 (a) and (b).



**Fig. 5.52(a)** Isolation Using Transformers

**Fig. 5.52(b)** Isolation Using Opto-Isolators

## SUMMARY

In this chapter, we have presented some of the peripheral devices and their interfacing circuits. To start with, the interfacing of static and dynamic memories have been discussed with examples. Then general concepts of I/O devices and their interfacing with 8086 are briefly explained. The parallel programmable peripheral interface 8255 has been presented in significant details along with the interfacing examples and programs for each of its operating modes. Further, a few important devices like ADCs, DACs and stepper motor have been discussed with an emphasis on their interfacing with 8086. The discussion concludes with a brief note on control of power (control) electronics devices. A number of advanced I/O devices are available which can be interfaced with the CPU using I/O ports, and the interfacing techniques of all these devices to be interfaced using I/O ports are similar in principle. Besides this, a large family of special purpose programmable peripheral devices is also available. This contributes considerably to the development of the recently available advanced computer, communication and automatic control systems. Some of these peripherals are explained in the next chapter.

## Exercises

- 5.1 Interface four 8K chips of static RAM and four 4K chips of EPROM with 8086. Interface two of the RAM chips in the zeroth segment so as to accommodate IVT in them. The remaining two RAM chips are to be interfaced at the end of the fifth segment. Two EPROM chips should be interfaced so that the system is restartable as usual, and the remaining two EPROM chips should be interfaced at the starting of A000th segment. Use absolute decoding scheme.
- 5.2 Interface eight 8K chips of RAM and four 8K chips of EPROM with 8086. Interface the RAM bank at a segment address 0B00H and the EPROM bank at a physical address F8000H. Do not allow any fold back space.
- 5.3 Interface eight 8K chips of RAM and four 8K × 4 chips of EPROM with 8086 at the further specified address map. You may use linear decoding scheme for minimising the required hardware.

Chips	Segment Address	Starting Offset Address
RAM1 and RAM2	0000H	0000H
RAM3 and RAM4	0500H	5000H
RAM5 and RAM6	7000H	2000H
RAM7 and RAM8	E000H	A000H
EPROM1 and EPROM2	F000H	0000H
EPROM3 and EPROM4	D000H	7000H

Will this system be practically useful? Explain. If not then what minimum change do you suggest in this address map?

- 5.4 Interface two 8K RAM chips and two 4K EPROM chips with 8088 so as to form a completely working system configuration.
- 5.5 Describe the procedure of interfacing static memories with a CPU? Bring out the differences between interfacing the memories with 8086 and 8088.
- 5.6 Bring out the differences between static and dynamic RAM.
- 5.7 Design a 2-digit seconds counter using 74373 output ports.
- 5.8 Design a 3-digit pulse counter using 74373 output ports to count TTL compatible pulses using an input line of a 74245 input port.
- 5.9 Generate a square wave of period 1sec using 74373 output ports.
- 5.10 Design a multiplexed display scheme to display seconds, minutes and hours counter using 8255 ports. Assume that a standard delay of 1 second is available.
- 5.11 Design a one unit 14-segment alphanumeric display and write a program to display an alphanumeric character of which the code is in AX.
- 5.12 Interface an 8255 with 8086 so as to have port A address 00, port B address 02, port C address 01 and CWR address 03.
- 5.13 Explain the different modes of operation of 8255.
- 5.14 Explain the control word format of 8255 in I/O and BSR mode.
- 5.15 Write a program to print message—'This is a printer test routine.' to a printer using 8255 port initialised in mode1.
- 5.16 Interface an 8'8 keyboard using two 8255 ports and write a program to read the code of a pressed key.
- 5.17 Interface a typical 12-bit DAC with 8255 and write a program to generate a triangular waveform of period 10 ms. The CPU runs at 5 MHz clock frequency.

- 5.18 Using a typical 12-bit DAC generate a step waveform of duration 1sec, maximum voltage 3 volts and determine duration of each step suitably.
- 5.19 Draw and discuss analog and digital section of 7109 in brief.
- 5.20 Design a 7109 circuit to convert the analog voltage to digital equivalent at a rate of 30 samples per second.
- 5.21 Draw a typical stepper motor interface with 8255.
- 5.22 Write ALPs to rotate a 200 teeth, 4 phase stepper motor as specified below.
  - (i) Five rotations clockwise and then five rotations anticlockwise.
  - (ii) Rotate through angle  $135^\circ$  in 2 sec.
  - (iii) Rotate the shaft at a speed of 10 rotations per minute.
- 5.23 Write ALPs to trigger a triac with a +5 V pulse of 200 msec as specified below.
  - (i) At an angle  $45^\circ$  in each positive and negative half cycle.
  - (ii) At an angle  $30^\circ$  in each positive half cycle.
  - (iii) At an angle  $20^\circ$  in each negative half.

Assume that after each zero crossing of a 50 Hz line signal an external zero crossing circuit generates an interrupt to the CPU which runs at a frequency of 5 MHz.

# Special Purpose Programmable Peripheral Devices and Their Interfacing

## INTRODUCTION

In the previous chapter, we discussed a few general purpose peripheral devices along with their interfacing techniques and example programs. More or less all the peripheral devices provide interface between the CPU and slow electromechanical processes or devices. Previously, all these interfaces were taken care of by the CPU using interrupt mechanism or polling techniques. However, due to the low speed of the processes a considerable amount of CPU time gets consumed in the interface and communication activities, resulting in reduced overall efficiency and low processing speed. To minimize the slow speed I/O communication overhead, a set of dedicated programmable peripheral devices have been introduced. Once initiated by the CPU, these programmable peripheral devices take care of all the interface activities for which they have been designed. Thus the CPU becomes free from the interface activities and can execute its main task more efficiently. In this chapter, we present several integrated programmable peripherals and their interfacing techniques. More advanced peripherals based on DMA controlled data transfer will be discussed in the next chapter.

### 6.1 PROGRAMMABLE INTERVAL TIMER 8253

In Chapter 4, we have shown that it is not possible to generate an arbitrary time delay precisely using delay routines. Intel's programmable counter/timer device ( 8253 ) facilitates the generation of accurate time delays. When 8253 is used as a timing and delay generation peripheral, the microprocessor becomes

free from the tasks related to the counting process and can execute the programs in memory, while the timer device may perform the counting tasks. This minimizes the software overhead on the microprocessor.

### 6.1.1 Architecture and Signal Descriptions

The programmable timer device 8253 contains three independent 16-bit counters, each with a maximum count rate of 2.6 MHz. It is thus possible to generate three totally independent delays or maintain three independent counters simultaneously. All the three counters may be independently controlled by programming the three internal command word registers.

The 8-bit, bidirectional data buffer interfaces internal circuit of 8253 to microprocessor systems bus. Data is transmitted or received by the buffer upon the execution of IN or OUT instruction. The read/write logic controls the direction of the data buffer depending upon whether it is a read or a write operation. It may be noted that IN instruction reads data while OUT instruction writes data to a peripheral. The internal block diagram and pin diagram of 8253 are shown in Figs 6.1(a) and 6.1(b), respectively.

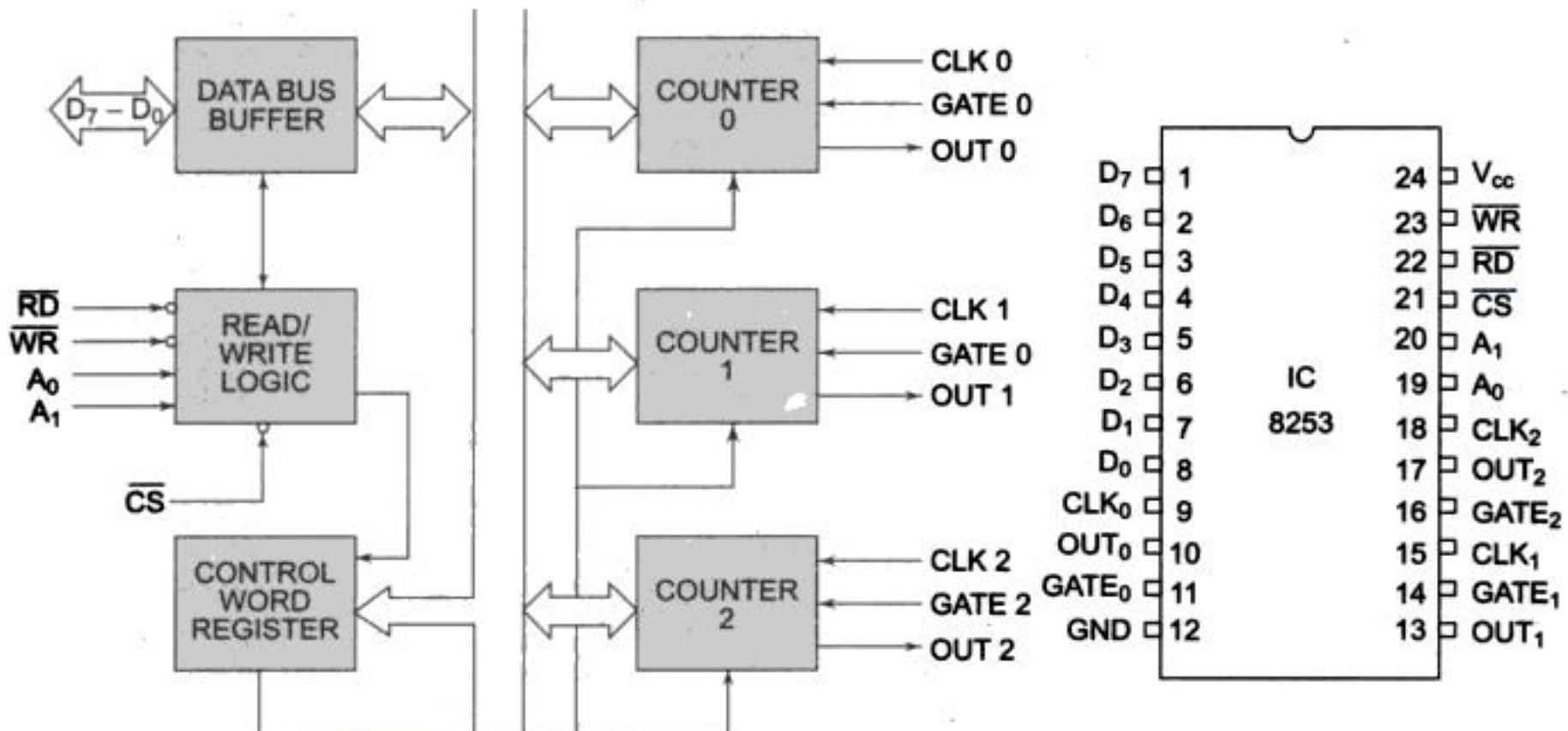


Fig. 6.1(a) Internal Block Diagram of 8253

Fig. 6.1(b) Pin Configuration of 8253

The three counters available in 8253 are independent of each other in operation, but they are identical to each other in organization. These are all 16-bit presetable, down counters, able to operate either in BCD or in hexadecimal mode. The mode control word register contains the information that can be used for writing or reading the count value into or from the respective count register using the OUT and IN instructions. The speciality of the 8253 counters is that they can be easily read on line without disturbing the clock input to the counter. This facility is called as "on the fly" reading of counters, and is invoked using a mode control word.

A<sub>0</sub>, A<sub>1</sub> pins are the address input pins and are required internally for addressing the mode control word registers and the three counter registers. A low on 'CS' line enables the 8253. No operation will be performed by 8253 till it is enabled. Table 6.1 shows the selected operations for various control inputs.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
SC <sub>1</sub>	SC <sub>0</sub>	RL <sub>1</sub>	RL <sub>0</sub>	M <sub>2</sub>	M <sub>1</sub>	M <sub>0</sub>	BCD

Control Word Format

SC <sub>1</sub>	SC <sub>0</sub>	OPERATION
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Illegal

SC-Select Counter Bit Definitions

RL <sub>1</sub>	RL <sub>0</sub>	OPERATION
0	0	Latch Counter for 'ON THE FLY' reading
0	1	Read/Load Least Significant Byte only
1	0	Read/Load MSB only
1	1	Read/Load LSB first then MSB

RL-Read/Load Bit Definitions

M <sub>2</sub>	M <sub>1</sub>	M <sub>0</sub>	Selected Mode
0	0	0	Mode 0
0	0	1	Mode 1
x	1	0	Mode 2
x	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

M<sub>2</sub>M<sub>1</sub>M<sub>0</sub> Mode Select Bit Definitions

BCD	Operation
0	Hexadecimal Count
1	BCD Count

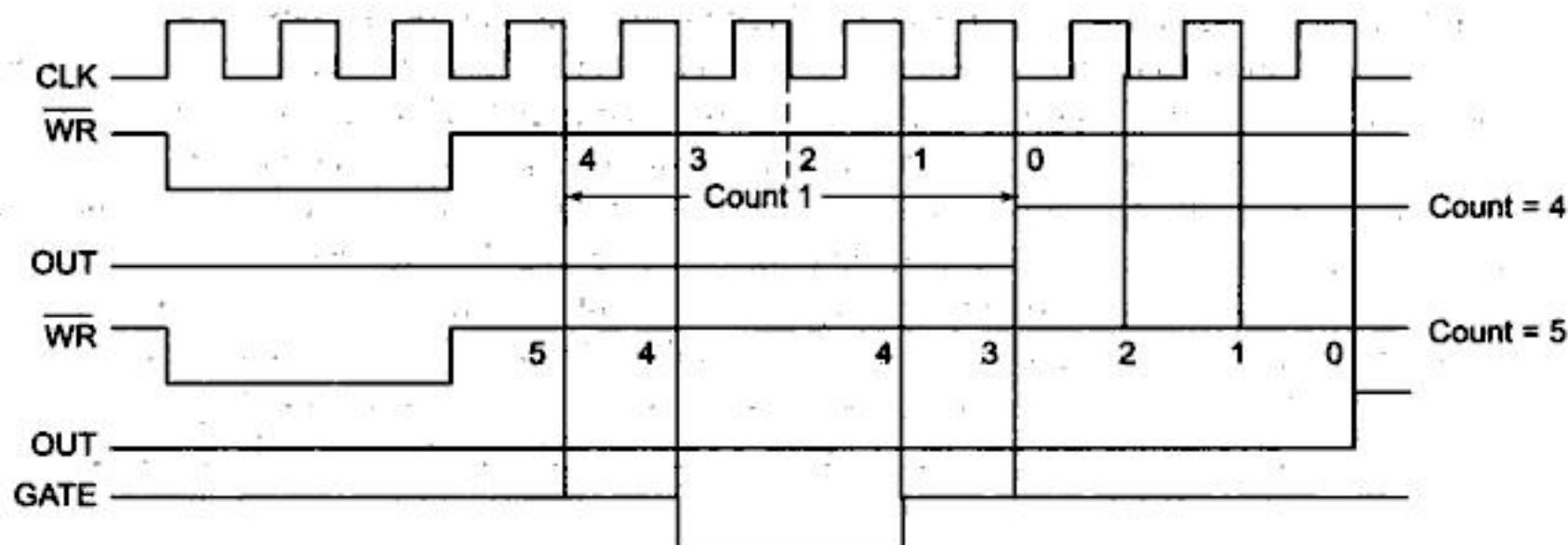
HEX/BCD Bit Definition

Fig. 6.2 Control Word Format and Bit Definitions

**MODE 0** This mode of operation is generally called as *interrupt on terminal count*. In this mode, the output is initially low after the mode is set. The output remains low even after the count value is loaded in the counter. The counter starts decrementing the count value after the falling edge of the clock, if the GATE input is high. The process of decrementing the counter continues at each falling edge of the clock till the terminal count is reached, i.e. the count becomes zero. When the terminal count is reached, the output goes high and remains high till the selected control word register or the corresponding count register is reloaded with a new mode of operation or a new count, respectively. This high output may be used to interrupt the processor whenever required, by setting a suitable terminal count. Writing a count register while the previous counting is in process, generates the following sequence of response.

The first byte of the new count when loaded in the count register, stops the previous count. The second byte when written, starts the new count, terminating the previous count then and there.

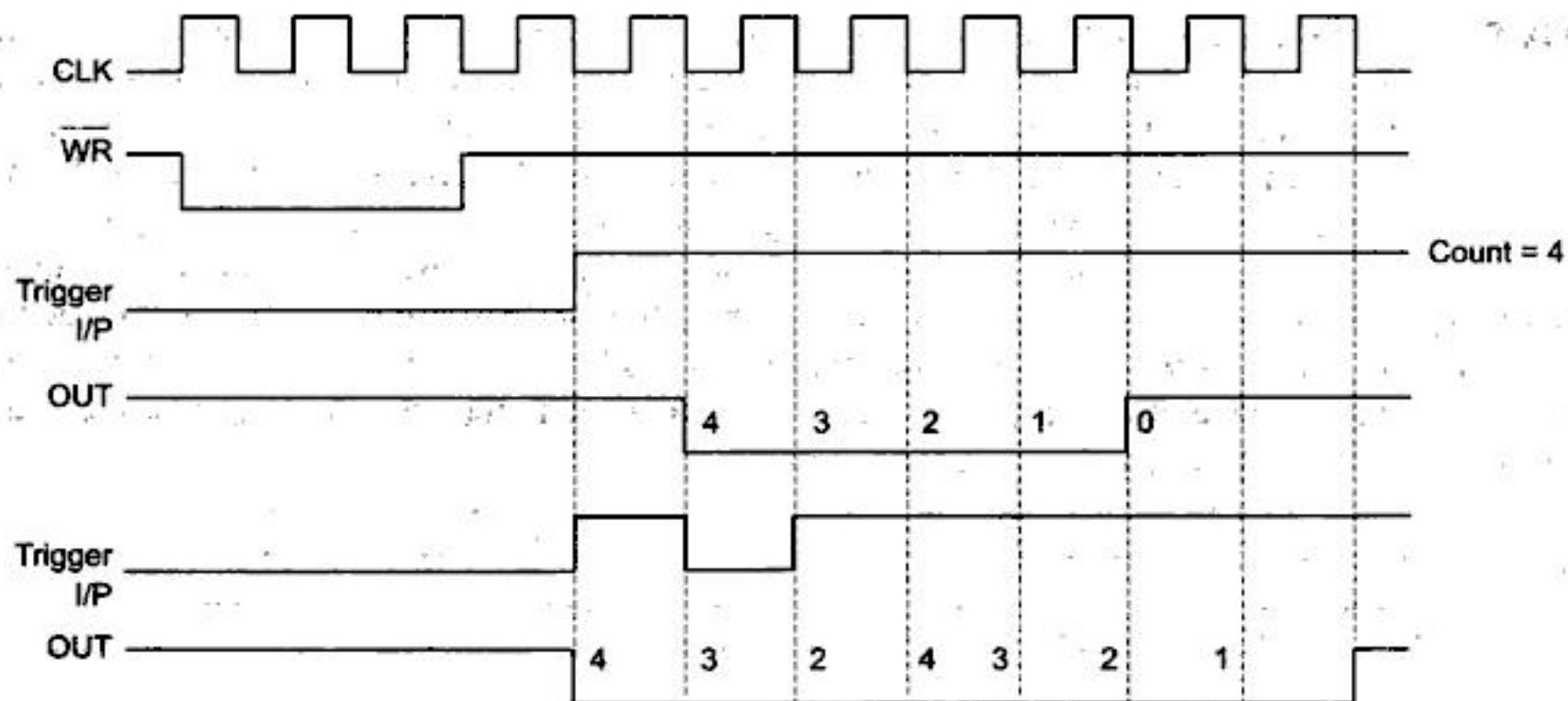
The GATE signal is active high and should be high for normal counting. When GATE goes low counting is terminated and the current count is latched till the GATE again goes high. Figure 6.3 shows the operational waveforms in mode 0.



**Fig. 6.3** Waveforms of  $\overline{WR}$ , OUT and GATE in Mode 0

**MODE 1** This mode of operation of 8253 is called as *programmable one-shot mode*. As the name implies, in this mode, the 8253 can be used as a *monostable multivibrator*. The duration of the quasistable state of the monostable multivibrator is decided by the count loaded in the count register. The gate input is used as trigger input in this mode of operation. Normally the output remains high till the suitable count is loaded in the count register and a trigger is applied. After the application of a trigger (on the positive edge), the output goes low and remains low till the count becomes zero. If another count is loaded when the output is already low, it does not disturb the previous count till a new trigger pulse is applied at the GATE input. The new counting starts after the new trigger pulse. Figure 6.4 shows the related waveforms for mode 1.

**MODE 2** This mode is called either *rate generator* or *divide by N counter*. In this mode, if  $N$  is loaded as the count value, then, after  $N$  pulses, the output becomes low only for one clock cycle. The



**Fig. 6.4**  $\overline{WR}$  GATE OUT Waveforms for Mode 1



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

    OUT 44H, AL      ; into count register
    MOV AH, 4CH      ; Stop
    INT2IH
CODE
ENDS
END START

```

**Program 6.3 ALP for Problem 6.1 (c)****Problem 6.2**

Interface 8253 with 8086 at counter 0 address 7430 H and write a program to call subroutine after 100 ms. Assume that the system clock available is 2 MHz.

**Solution**

Address Decoding table for 8253 PIT.

<b>Port</b>	<b>HEX. address</b>	<b>Binary address</b>							
		<b>A<sub>15</sub></b>	-----	<b>A<sub>3</sub></b>	<b>A<sub>2</sub></b>	<b>A<sub>1</sub></b>	<b>A<sub>0</sub></b>		
Counter 0	7430 H	0111	0100	0011	0	0	0	0	
Counter 1	7432 H	0111	0100	0011	0	0	1	0	
Counter 2	7434 H	0111	0100	0011	0	1	0	0	
CWR	7436 H	0111	0100	0011	0	1	1	0	

**Description:**

1. Input circuit signal to counter 0 is 2 MHz, Counter 0 is utilized in mode 3 to generate a square wave of 1 kHz.
2. Output square wave of counter 0 is given as circuit signal to counter 1, counter 1 is utilized in mode 0,  $t_d$  i.e. interrupt on terminal count.

Counter 1: For generating hardware delay of 100 ms counter 1 is used in mode 0 (i.e. interrupt on terminal count).

$$t_d = n \times t_c$$

where  $t_d$  = delay time  
 $n$  = no. of count loaded in counter 1  
 $t_c$  = time period of applied circuit

$$\therefore t_c = \frac{1}{1\text{kHz}} = 1 \text{ ms}$$

Given  $t_d = 100 \text{ ms}$ .

$$\therefore n = \frac{t_d}{t_c} = \frac{100 \text{ ms}}{1\text{ms}} = 00$$

$$n = (100)_{\text{Decimal}}$$

Counter 0: For generating a square wave of 1 kHz counter 0 is used in mode 3 (i.e. square wave generation mode).

In mode 3

$$n = \frac{\text{input frequency}}{\text{output frequency}} = \frac{2 \text{ MHz}}{1 \text{ kHz}} = \frac{2000 \text{ kHz}}{1 \text{ kHz}}$$

$$\therefore n = (2000)_{\text{Decimal}}$$

2. Output square wave of counter 0 is given as Clk signal to Counter 1, Counter 1 is utilized in mode 0, i.e. interrupt on terminal count.

Counter 1: For generating hardware delay of 100 msec Counter 1 is used in mode 0 (i.e. interrupt on terminal count).

Where

$$t_d = n \times t_c$$

$t_d$  = delay time

$n$  = count loaded in Counter 1

$t_c$  = time period of applied circuit.

$\therefore$

$$t_c = \frac{1}{1 \text{ kHz}} = 1 \text{ ms.}$$

Given

$$t_d = 100 \text{ ms}$$

$\therefore$

$$n = \frac{t_d}{t_c} = \frac{100 \text{ ms}}{1 \text{ ms}} = 100$$

$\therefore$

$$n = (100)_{\text{Decimal}}$$

CWR:

SC 1	SC 0	RL 1	RL 0	M 2	M 1	M 0	BCD
------	------	------	------	-----	-----	-----	-----

For Counter 0:

0	0	1	0	0	1	1	1	→ 27 H
---	---	---	---	---	---	---	---	--------

Counter 0      Loading      Mode 3      Counter as a BCD,

Only MSB

$$\therefore n = (2000)_{\text{Decimal}}$$

We have to load only MSB, i.e.  $(20)_D$ , whereas Counter is automatically initialized to  $(00)_D$

For Counter 1:

0	1	1	0	0	0	0	1	→ 61 H
---	---	---	---	---	---	---	---	--------

Counter 1      Loading      Mode 0      BCD Counter.

only MSB

$$n = (0100)_D$$

$\therefore$  We have to load only MSB, i.e.  $(01)_D$

ASSUME CS : CODE, SS : STACK

STACK SEGMENT

TOP DW 100 DUP (?)

STACK ENDS

CODE SEGMENT

Start : MOV AX, STACK

MOV SS, AX

LEA SP, TOP + 200

```

MOV DX,7436 H ; CWR address is transferred to DX register.
MOV AL,27H ; Initialization of counter 0.
OUT DX,AL
MOV AL,61H ; Initialization of counter 1
OUT DX,AL
MOV AL,20H
MOV DX,7430H ; Count 20 loaded in BCD counter 0 MSBs.
OUT DX,AL
MOV AL,01H
MOV DX,7432 H ; Count 01 is loaded in MSBs of counter 1
OUT DX,AL
STI ; Set IF Flag.
MOV AH,4 CH
INT 21H
CODE ENDS
END Start

```

Program 6.4 ALP for Problem 6.2

The circuit arrangement for this interfacing problem is as shown below:

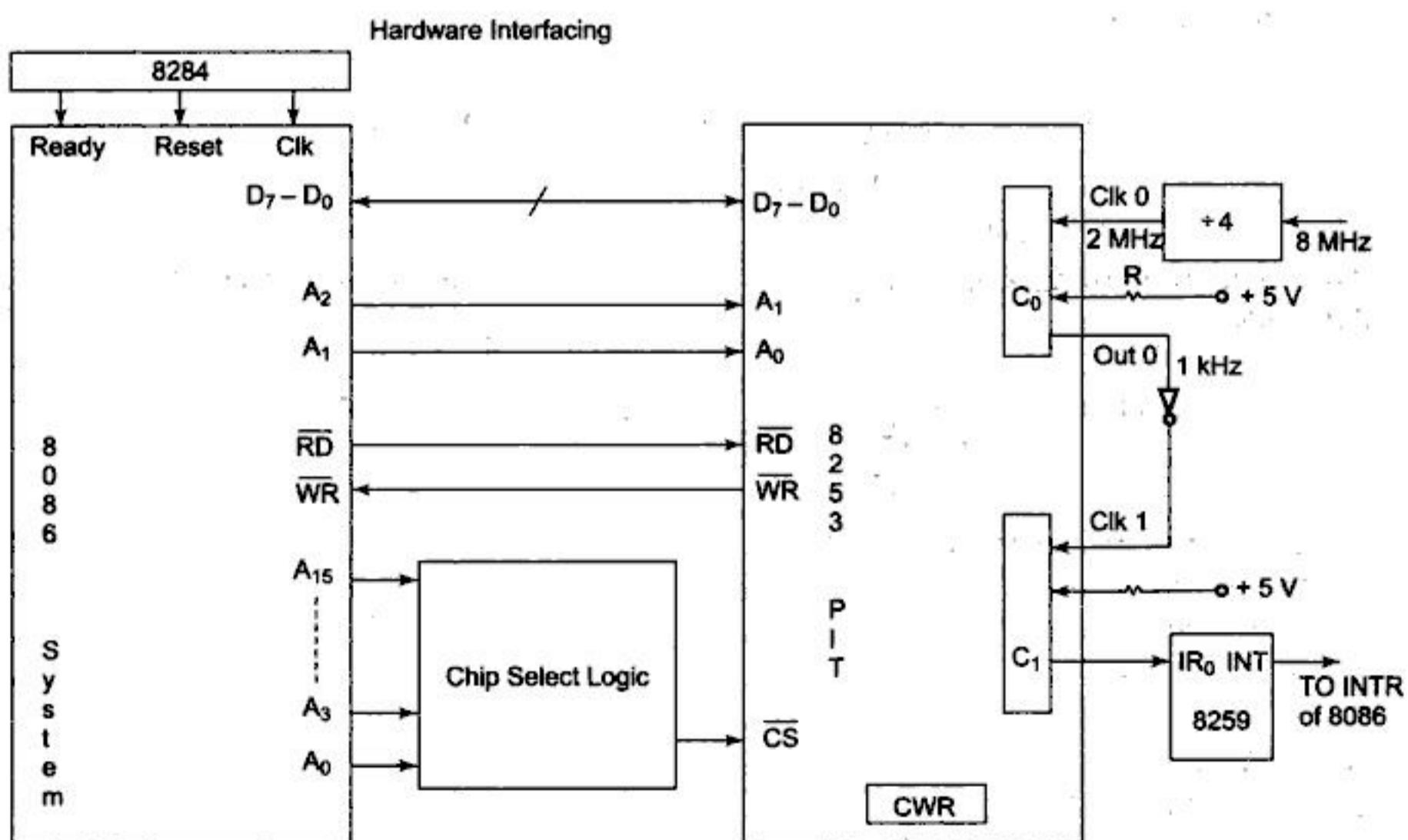


Fig. 6.11 Interfacing 8253 with 8086 for Problem 6.2

In all the above programs, the counting down starts as soon as the writing operation to the count register is over, and the WR pin goes high after writing. In case of 'read on fly' operation, the READ ON FLY control word for the specific counter is to be loaded in the control word register followed by the successive read operations. The 8253 and 8254 are the most widely used peripherals to generate accurate delays for industrial or laboratory purposes. The 8254 is similar to 8253 in architecture, programming and operation, hence 8254 is not discussed in this text.

## 6.2 PROGRAMMABLE INTERRUPT CONTROLLER 8259A

The processor 8085 had five hardware interrupt pins. Out of these five interrupt pins, four pins were allotted fixed vector addresses but the pin INTR was not allotted any vector address, rather an external device was supposed to hand over the type of the interrupt, i.e. (Type 0 to 7 for RST0 to RST7), to the microprocessor. The microprocessor then gets this type and derives the interrupt vector address from that. Consider an application, where a number of I/O devices connected with a CPU desire to transfer data using interrupt driven data transfer mode. In these types of applications, more number of interrupt pins are required than available in a typical microprocessor. Moreover, in these multiple interrupt systems, the processor will have to take care of the priorities for the interrupts, simultaneously occurring at the interrupt request pins.

To overcome all these difficulties, we require a programmable interrupt controller which is able to handle a number of interrupts at a time. This controller takes care of a number of simultaneously appearing interrupt requests along with their types and priorities. This relieves the processor from all these tasks. The programmable interrupt controller 8259A from Intel is one such device. Its predecessor 8259 was designed to operate only with 8-bit processors like 8085. A modified version, 8259A was later introduced that is compatible with 8-bit as well as 16-bit processors.

### 6.2.1 Architecture and Signal Descriptions of 8259A

The architectural block diagram of 8259A is shown in Fig. 6.12. The functional explanation of each block is given in the following text in brief:

**Interrupt Request Register (IRR)** The interrupts at IRQ input lines are handled by Interrupt Request Register internally. IRR stores all the interrupt requests in it in order to serve them one by one on the priority basis.

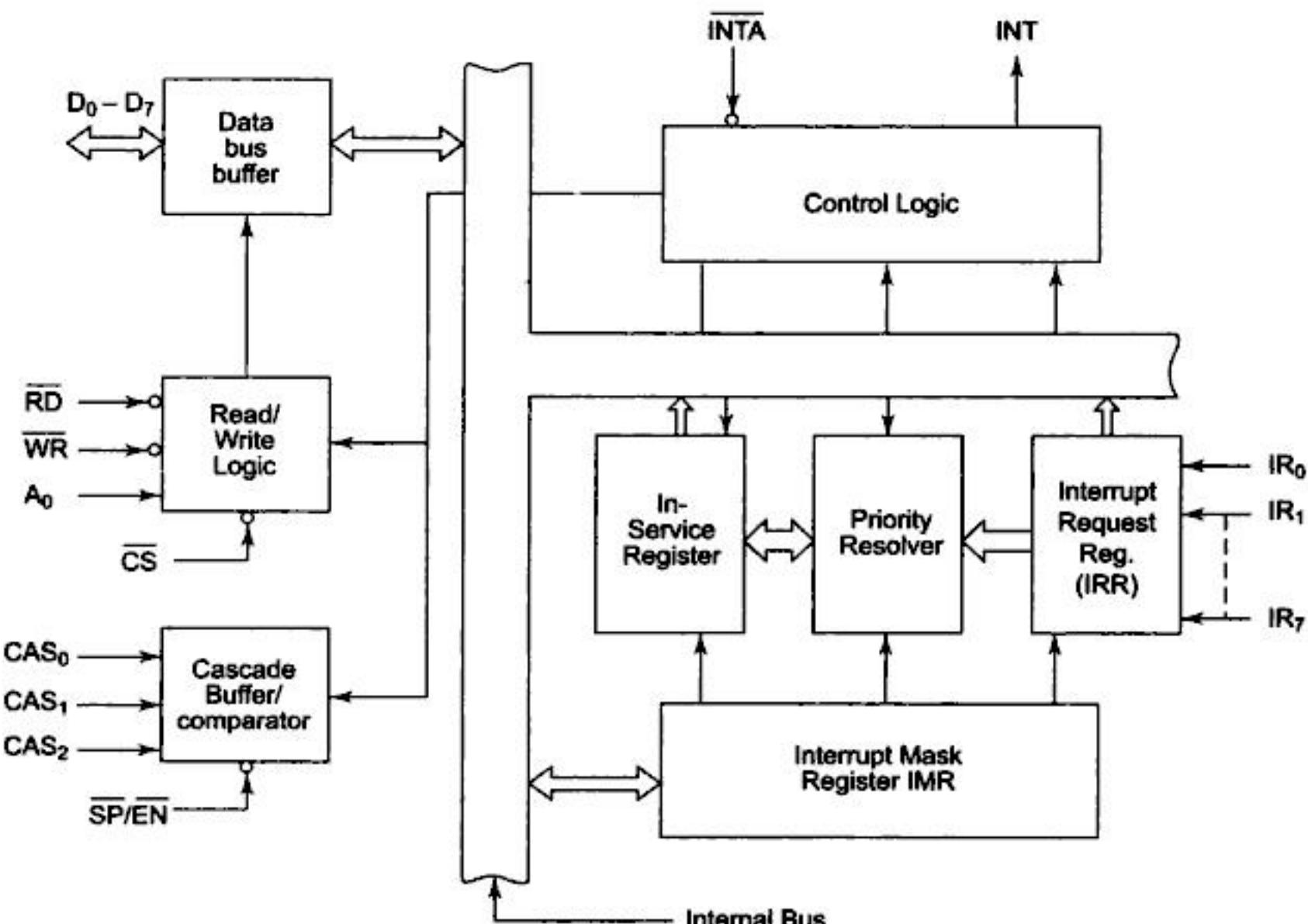
**In-Service Register (ISR)** This stores all the interrupt requests those are being served, i.e. ISR keeps a track of the requests being served.

**Priority Resolver** This unit determines the priorities of the interrupt requests appearing simultaneously. The highest priority is selected and stored into the corresponding bit of ISR during INTA pulse. The IR<sub>0</sub> has the highest priority while the IR<sub>7</sub> has the lowest one, normally in fixed priority mode. The priorities however may be altered by programming the 8259A in rotating priority mode.

**Interrupt Mask Register (IMR)** This register stores the bits required to mask the interrupt inputs. IMR operates on IRR at the direction of the Priority Resolver.

**Interrupt Control Logic** This block manages the interrupt and the interrupt acknowledge signals to be sent to the CPU for serving one of the eight interrupt requests. This also accepts the interrupt

acknowledge (INTA) signal from CPU that causes the 8259A to release vector address on to the data bus.



**Fig. 6.12 8259A Block Diagram**

**Data Bus Buffer** This tristate bidirectional buffer interfaces internal 8259A bus to the microprocessor system data bus. Control words, status and vector information pass through data buffer during read or write operations.

**Read/Write Control Logic** This circuit accepts and decodes commands from the CPU. This block also allows the status of the 8259A to be transferred on to the data bus.

**Cascade Buffer/Comparator** This block stores and compares the IDs of all the 8259As used in the system. The three I/O pins CAS0-2 are outputs when the 8259A is used as a master. The same pins act as inputs when the 8259A is in the slave mode. The 8259A in the master mode, sends the ID of the interrupting slave device on these lines. The slave thus selected, will send its pre-programmed vector address on the data bus during the next INTA pulse.

Figure 6.13 shows the pin configuration of 8259A, followed by their functional description of each of the signals in brief.

**CS** This is an active-low chip select signal for enabling  $\overline{RD}$  and  $\overline{WR}$  operations of 8259A.  $\overline{INTA}$  function is independent of  $\overline{CS}$ .

**WR** This pin is an active-low write enable input to 8259A. This enables it to accept command words from CPU.

**RD** This is an active-low read enable input to 8259A. A low on this line enables 8259A to release status onto the data bus of CPU.

**D<sub>7</sub>-D<sub>0</sub>** These pins form a bidirectional data bus that carries 8-bit data either to control word or from status word registers. This also carries interrupt vector information.

**CAS<sub>0</sub>-CAS<sub>2</sub>, Cascade Lines** A single 8259A provides eight vectored interrupts. If more interrupts are required, the 8259A is used in the cascade mode in which a master 8259A along with eight slaves 8259A can provide up to 64 vectored interrupt lines. These three lines act as select lines for addressing the slaves 8259A.

**PS/EN** This pin is a dual purpose pin. When the chip is used in buffered mode, it can be used as a buffer enable to control buffer transreceivers. If this is not used in buffered mode then the pin is used as input to designate whether the chip is used as a master ( $\overline{SP} = 1$ ) or a slave ( $\overline{EN} = 0$ ).

**INT** This pin goes high whenever a valid interrupt request is asserted. This is used to interrupt the CPU and is connected to the interrupt input of CPU.

**IR<sub>0</sub>-IR<sub>7</sub>,(Interrupt requests)** These pins act as inputs to accept interrupt requests to the CPU. In the edge triggered mode, an interrupt service is requested by raising an IR pin from a low to a high state. It is held high until it is acknowledged, and just by latching it to high level, if used in the level triggered mode.

**INTA (Interrupt acknowledge)** This pin is an input used to strobe-in 8259A interrupt vector data on to the data bus. In conjunction with **CS**, **WR**, and **RD** pins, this selects the different operations like, writing command words, reading status word, etc.

The device 8259A can be interfaced with any CPU using either polling or interrupt. In polling, the CPU keeps on checking each peripheral device in sequence to ascertain if it requires any service from the CPU. If any such service request is noticed, the CPU serves the request and then goes on to the next device in sequence. After all the peripheral devices are scanned as above the CPU again starts from the first device. This type of system operation results in the reduction of processing speed because most of the CPU time is consumed in polling the peripheral devices.

In the interrupt driven method, the CPU performs the main processing task till it is interrupted by a service requesting peripheral device. The net processing speed of these type of systems is high because

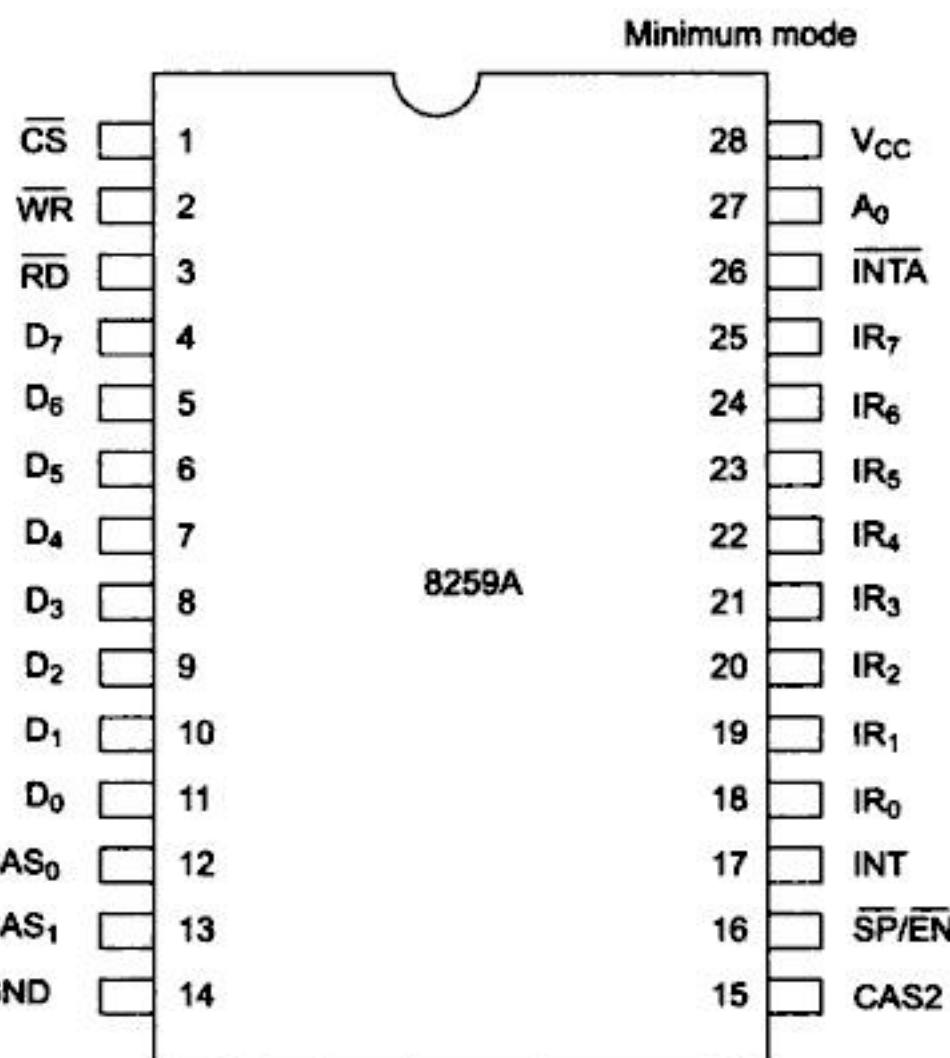


Fig. 6.13 8259 Pin Diagram

the CPU serves the peripheral only if it receives the interrupt request. If more than one interrupt requests are received at a time, all the requesting peripherals are served one by one on priority basis. This method of interfacing may require additional hardware if number of peripherals to be interfaced is more than the interrupt pins available with the CPU.

### 6.2.2 Interrupt Sequence in an 8086 System

The interrupt sequence in an 8086-8259A system is described as follows:

1. One or more IR lines are raised high that set corresponding IRR bits.
2. 8259A resolves priority and sends an INT signal to CPU.
3. The CPU acknowledges with  $\overline{\text{INTA}}$  pulse.
4. Upon receiving an  $\overline{\text{INTA}}$  signal from the CPU, the highest priority ISR bit is set and the corresponding IRR bit is reset. The 8259A does not drive data bus during this period.
5. The 8086 will initiate a second  $\overline{\text{INTA}}$  pulse. During this period 8259A releases an 8-bit pointer on to data bus from where it is read by the CPU.
6. This completes the interrupt cycle. The ISR bit is reset at the end of the second  $\overline{\text{INTA}}$  pulse if automatic end of interrupt (AOEI) mode is programmed. Otherwise ISR bit remains set until an appropriate EOI command is issued at the end of interrupt subroutine.

### 6.2.3 Command Words of 8259A

The command words of 8259A are classified in two groups, viz. Initialization Command Words (ICWs) and Operation Command Words (OCWs).

**Initialization Command Words (ICWs)** Before it starts functioning, the 8259A must be initialized by writing two to four command words into the respective command word registers. These are called as Initialization Command Words (ICWs). If  $A_0 = 0$  and  $D_4 = 1$ , the control word is recognized as  $ICW_1$ . It contains the control bits for edge/level triggered mode, single/cascade mode, call address interval and whether  $ICW_4$  is required or not, etc. If  $A_0 = 1$ , the control word is recognized as  $ICW_2$ . The  $ICW_2$  stores details regarding interrupt vector addresses. The initialization sequence of 8259A is described in form of a flow chart in Fig. 6.14. The bit functions of the  $ICW_1$  and  $ICW_2$  are self explanatory as shown in Fig. 6.15.

Once  $ICW_1$  is loaded, the following initialization procedure is carried out internally.

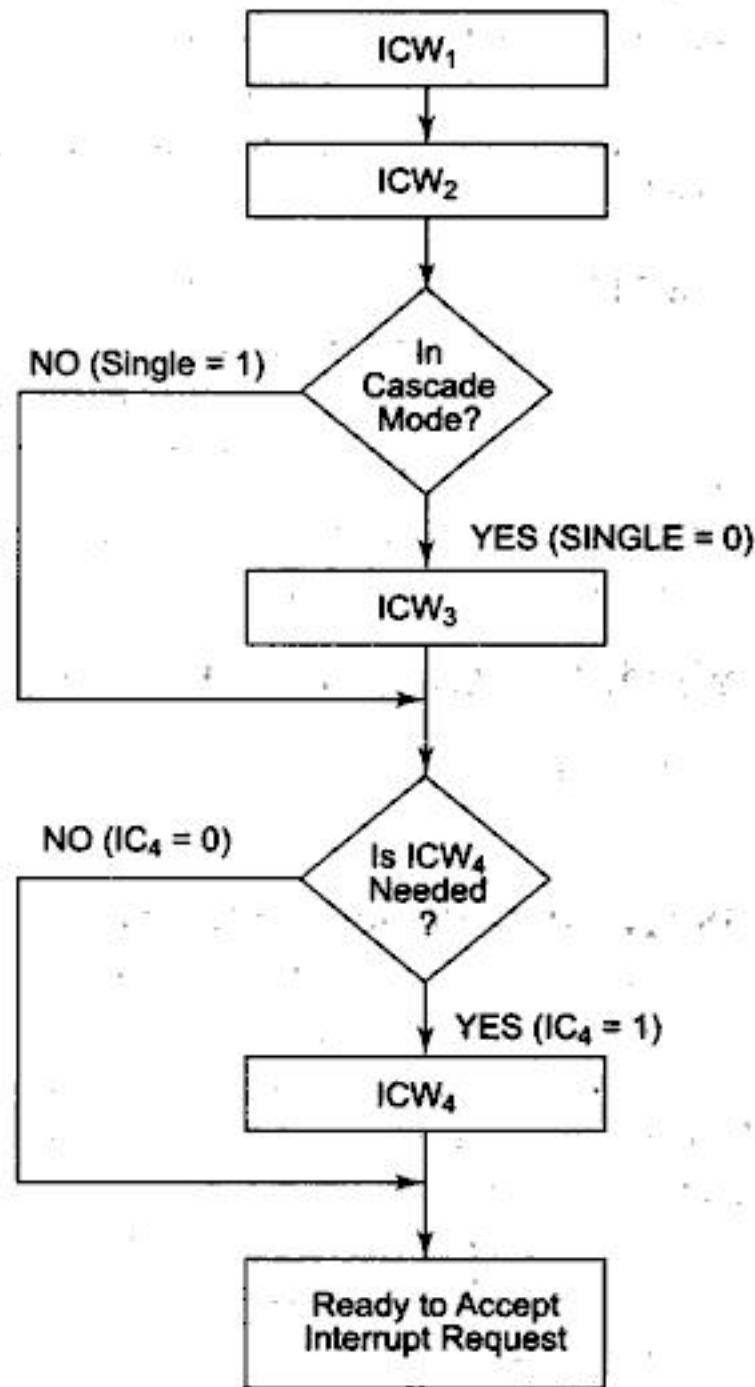


Fig. 6.14 Initialization Sequence of 8259A

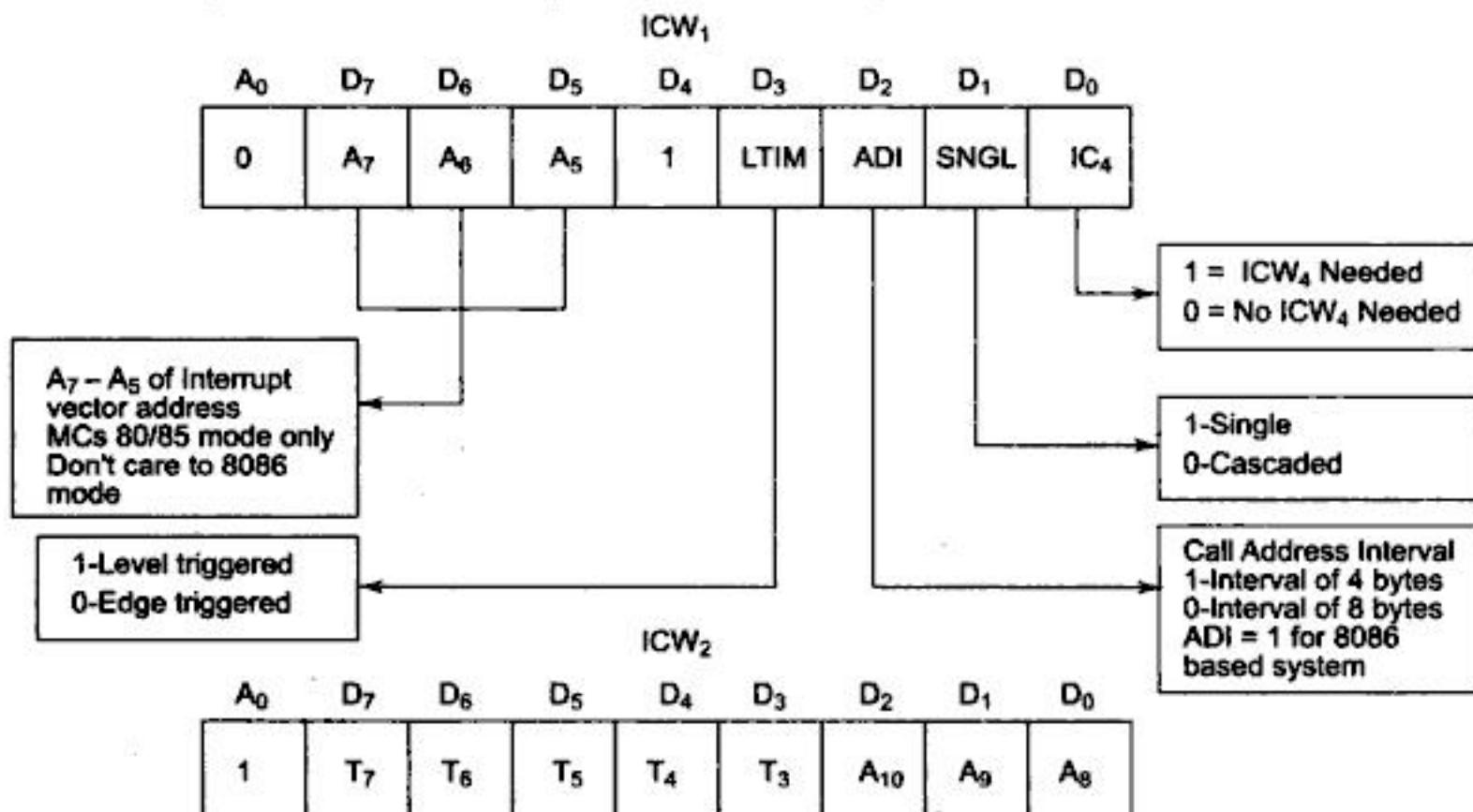
- (a) The edge sense circuit is reset, i.e. by default 8259A interrupts are edge sensitive
  - (b) IMR is cleared
  - (c) IR7 input is assigned the lowest priority
  - (d) Slave mode address is set to 7
  - (e) Special mask mode is cleared and the status read is set to IRR
  - (f) If  $IC_4 = 0$ , all the functions of  $ICW_4$  are set to zero. Master/slave bit in  $ICW_4$  is used in the buffered mode only.

In an 8085 based system,  $A_{15} - A_8$  of the interrupt vector address are the respective bits of ICW<sub>2</sub>. In 8086/88 based system, five most significant bits of the interrupt type byte are inserted in place of  $T_7 - T_3$  respectively and the remaining three bits ( $A_8, A_9$  and  $A_{10}$ ) are inserted internally as 000 (as if they are pointing to  $IR_0$ ). Other seven interrupt levels' vector addresses are internally generated automatically by 8259 using  $IR_n$  vector. Address interval is always four in an 8086 based system.

$ICW_1$  and  $ICW_2$  are compulsory command words in initialization sequence of 8259A as is evident from Fig. 6.14, while  $ICW_3$  and  $ICW_4$  are optional. The  $ICW_3$  is read only when there are more than one 8259As in the system, i.e. cascading is used ( $SNGL = 0$ ). The  $SNGL$  bit in  $ICW_1$  indicates whether the 8259A is in the cascade mode or not. The  $ICW_3$  loads an 8-bit slave register. Its detailed functions are as follows:

In the master mode (i.e.  $\overline{SP} = 1$  or in buffer mode  $M/S = 1$  in ICW4), the 8-bit slave register will be set bit-wise to '1' for each slave in the system, as shown in Fig. 6.16. The requesting slave will then release the second byte of a CALL sequence.

In slave mode (i.e.  $\overline{SP} = 0$  or if  $BUF = 1$  and  $M/S = 0$  in ICW 4) bits  $D_2$  to  $D_0$  identify the slave, i.e. 000 to 111 for slave1 to slave8. The slave compares the cascade inputs with these bits and if they are equal, the second byte of the CALL sequence is released by it on the data bus.



**T<sub>7</sub> – T<sub>3</sub>**—For 8085 system they are filled by A<sub>15</sub>–A<sub>11</sub> of the interrupt vector address and the least significant 3 bits are same as the respective bits of vector address. For 8086 system they are filled by most significant 5 bits of interrupt type and the least significant 3 bits are 0, pointing to Ir<sub>0</sub>.

**Fig. 6.15 Initialization Command Words ICW<sub>1</sub> and ICW<sub>2</sub>**

**ICW<sub>4</sub>** The use of this command word depends on the IC<sub>4</sub> bit of ICW<sub>1</sub>. If IC<sub>4</sub> = 1, ICW<sub>4</sub> is used, otherwise it is neglected. The bit functions of ICW<sub>4</sub> are described as follows:

**SFNM** Special fully nested mode is selected, if SFNM = 1.

**BUF** If BUF = 1, the buffered mode is selected. In the buffered mode, SP/EN acts as enable output and the master/slave is determined using the M/S bit of ICW<sub>4</sub>.

**M/S** If M/S = 1, 8259A is a master. If M/S = 0, 8259A is a slave. If BUF = 0, M/S is to be neglected.

Master mode ICW <sub>3</sub>									
A <sub>0</sub>	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	
1	S <sub>7</sub>	S <sub>6</sub>	S <sub>5</sub>	S <sub>4</sub>	S <sub>3</sub>	S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	

Sn = 1-IRn Input has a slave  
= 0-IRn Input does not have a slave

Slave mode ICW <sub>3</sub>									
A <sub>0</sub>	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	
1	0	0	0	0	0	ID <sub>2</sub>	ID <sub>1</sub>	ID <sub>0</sub>	

D<sub>2</sub>D<sub>1</sub>D<sub>0</sub> - 000 to 111 for IR<sub>0</sub> to IR<sub>7</sub> or slave 1 to slave 8

Fig. 6.16 ICW<sub>3</sub> in Master and Slave Mode

**AEOI** If AEOI = 1, the automatic end of interrupt mode is selected.

**mPM** If the mPM bit is 0, the MCS-85 system operation is selected and if mPM = 1, 8086/88 operation is selected.

Figure 6.17 shows the ICW<sub>4</sub> bit positions:

ICW <sub>4</sub>									
A <sub>0</sub>	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	
1	0	0	0	SFNM	BUF	M/S	AEOI	mPM	

Fig. 6.17 ICW<sub>4</sub> Bit Functions

**Operation Command Words** Once 8259A is initialized using the previously discussed command words for initialisation, it is ready for its normal function, i.e. for accepting the interrupts but 8259A has its own ways of handling the received interrupts called as modes of operation. These modes of operations can be selected by programming, i.e. writing three internal registers called as *operation command word registers*. The data written into them (bit pattern) is called as *operation command words*. In the three operation command words OCW<sub>1</sub>, OCW<sub>2</sub> and OCW<sub>3</sub>, every bit corresponds to some operational feature of the mode selected, except for a few bits those are either '1' or '0'. The three operation command words are shown in Fig. 6.18 (a), (b) and (c) with the bit selection details. OCW<sub>1</sub> is used to mask the unwanted interrupt requests. If the mask bit is '1', the corresponding interrupt request is masked, and if it is '0', the request is enabled. In OCW<sub>2</sub> the three bits, viz. R, SL and EOI control the end of interrupt, the rotate mode and their combinations as shown in Fig. 6.18 (b). The three bits L<sub>2</sub>, L<sub>1</sub> and L<sub>0</sub> in OCW<sub>2</sub> determine the interrupt level to be selected for operation, if the SL bit is active, i.e. '1'. The details of OCW<sub>2</sub> are shown in Fig. 6.18(b).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

#### 6.2.4 Operating Modes of 8259

The different modes of operation of 8259A can be programmed by setting or resting the appropriate bits of the ICWs or OCWs as discussed previously. The different modes of operation of 8259A are explained in the following text:

**Fully Nested Mode** This is the default mode of operation of 8259A.  $IR_0$  has the highest priority and  $IR_7$  has the lowest one. When interrupt requests are noticed, the highest priority request amongst them is determined and the vector is placed on the data bus. The corresponding bit of ISR is set and remains set till the microprocessor issues an EOI command just before returning from the service routine or the AEOI bit is set. If the ISR (In Service) bit is set, all the same or lower priority interrupts are inhibited but higher levels will generate an interrupt, that will be acknowledged only if the microprocessor's Interrupt enable Flag (IF) is set. The priorities can afterwards be changed by programming the rotating priority modes.

**End of Interrupt (EOI)** The ISR bit can be reset either with AEOI bit of  $ICW_1$  or by EOI command, issued before returning from the interrupt service routine. There are two types of EOI commands specific and non-specific. When 8259A is operated in the modes that preserve fully nested structure, it can determine which ISR bit is to be reset on EOI. When non-specific EOI command is issued to 8259A it will automatically reset the highest ISR bit out of those already set.

When a mode that may disturb the fully nested structure is used, the 8259A is no longer able to determine the last level acknowledged. In this case a specific EOI command is issued to reset a particular ISR bit. An ISR bit that is masked by the corresponding IMR bit, will not be cleared by a non-specific EOI of 8259A, if it is in special mask mode.

**Automatic Rotation** This is used in the applications where all the interrupting devices are of equal priority. In this mode, an Interrupt Request (IR) level receives lowest priority after it is served while the next device to be served gets the highest priority in sequence. Once all the devices are served like this, the first device again receives highest priority.

**Automatic EOI Mode** Till  $AEOI = 1$  in  $ICW_4$ , the 8259A operates in AEOI mode. In this mode, the 8259A performs a non-specific EOI operation at the trailing edge of the last  $\overline{INTA}$  pulse automatically. This mode should be used only when a nested multilevel interrupt structure is not required with a single 8259A.

**Specific Rotation** In this mode a bottom priority level can be selected, using  $L_2$ ,  $L_1$  and  $L_0$  in  $OCW_2$  and  $R = 1$ ,  $SL = 1$ ,  $EOI = 0$ . The selected bottom priority fixes other priorities. If  $IR_5$  is selected as a bottom priority, then  $IR_5$  will have least priority and  $IR_4$  will have a next higher priority. Thus  $IR_6$  will have the highest priority. These priorities can be changed during an EOI command by programming the rotate on specific EOI command in  $OCW_2$ .

**Special Mask Mode** In the special mask mode, when a mask bit is set in  $OCW_1$ , it inhibits further interrupts at that level and enables interrupt from other levels, which are not masked.

**Edge and Level Triggered Mode** This mode decides whether the interrupt should be edge triggered or level triggered. If bit LTIM of  $ICW_1 = 0$ , they are edge triggered, otherwise the interrupts are level triggered.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

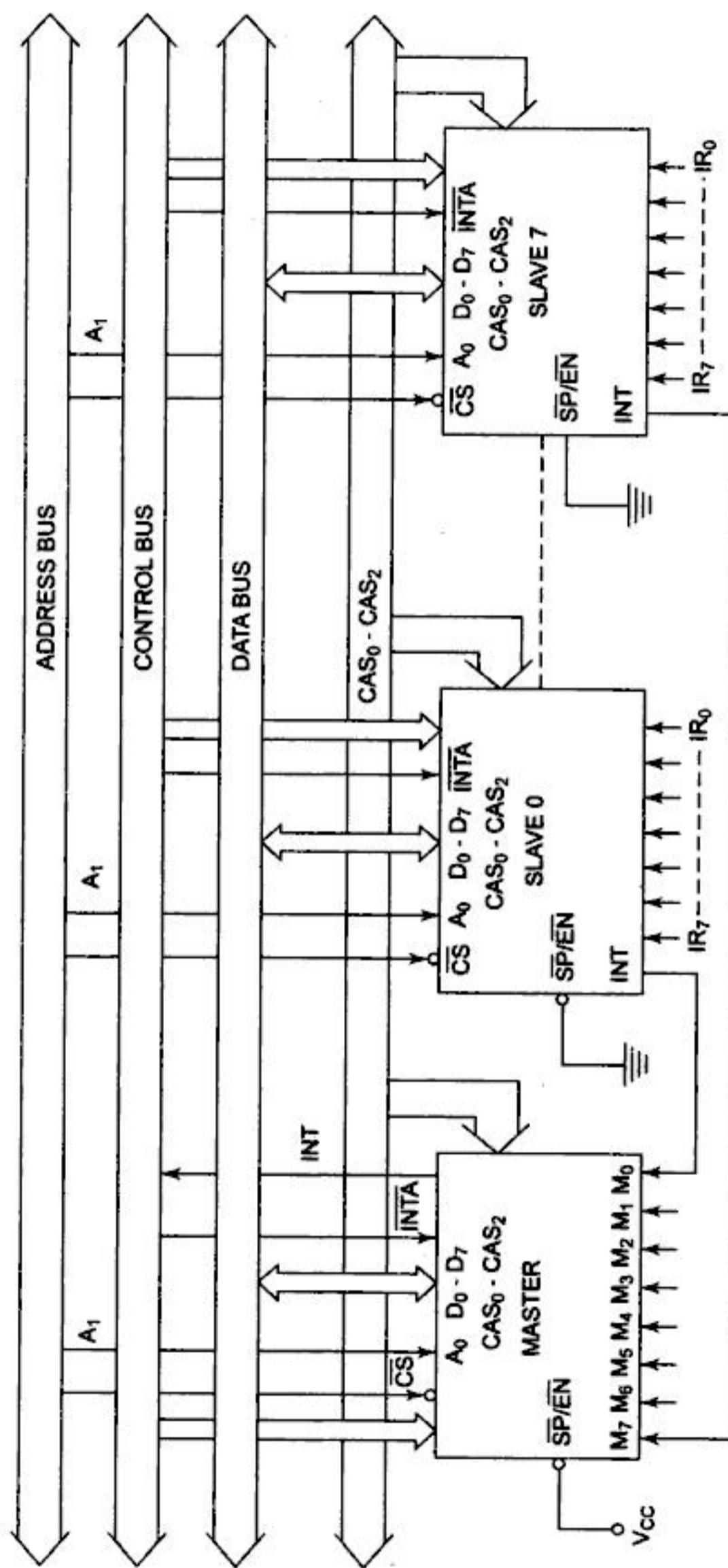


Fig. 6.20 8259A in Cascaded Mode

normally low and contain slave address codes from the trailing edge of the first INTA pulse to the trailing edge of the second INTA pulse. Each 8259A in the system must be separately initialized and programmed to work in different modes. The EOI command must be issued twice, one for master and the other for the slave. A separate address decoder is used to activate the chip select line of each 8259A. Figure 6.20 shows the details of the circuit connections of 8259As in cascade scheme.

### 6.2.5 Interfacing and Programming 8259

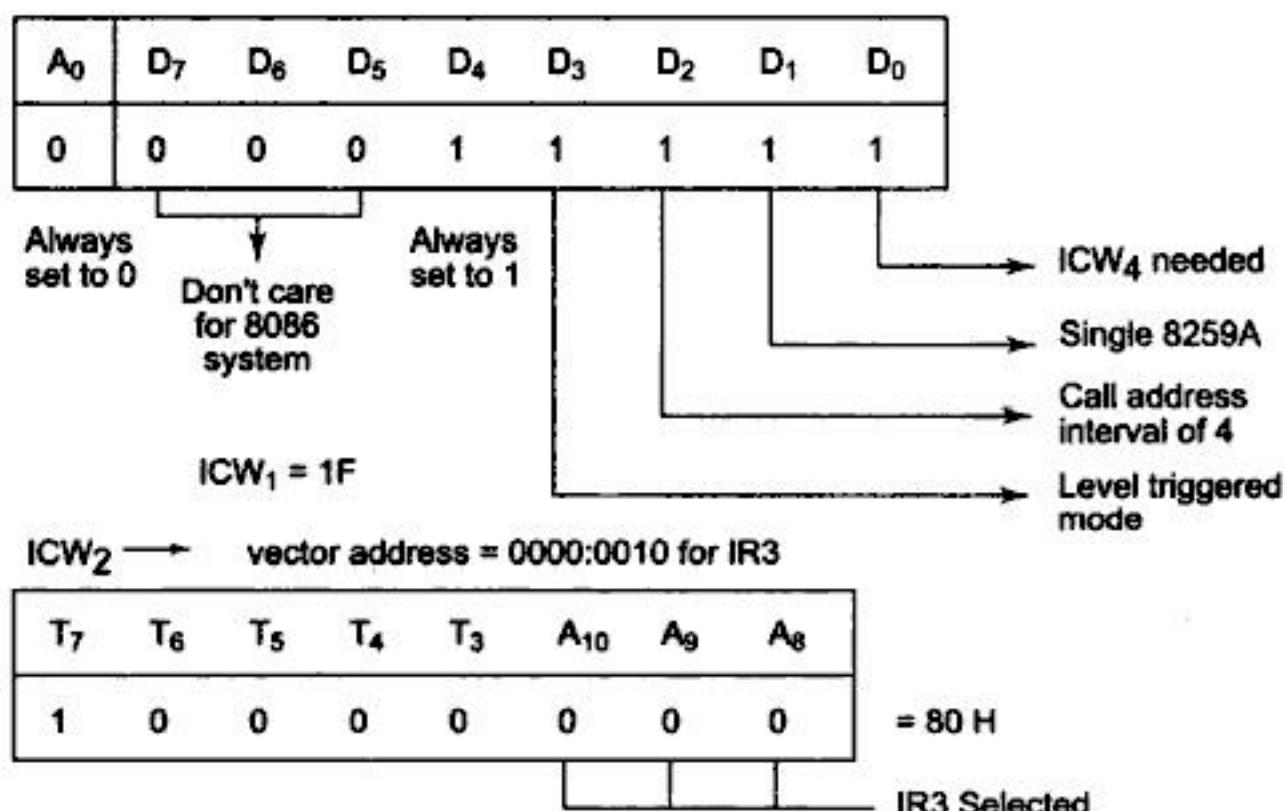
The following example elucidates the interfacing and programming of 8259 in an 8086 based system.

#### Problem 6.3

Show 8259A interfacing connections with 8086 at the address 074x. Write an ALP to initialize the 8259A in single level triggered mode, with call address interval of 4, non-buffered, no special fully nested mode. Then set the 8259A to operate with IR6 masked, IR4 as bottom priority level, with special EOI mode. Set special mask mode of 8259A. Read IRR and ISR into registers BH and BL respectively. IR<sub>0</sub> of 8259 will have type 80H.

#### Solution

Let the starting vector address is 0000:0200H ( 80H X 4 in segment 0000H ). The interconnections of 8259A with 8086 are shown in Fig. 6.21. The 8259 is interfaced with lower byte of the 8086 data bus, hence A<sub>0</sub> line of the microprocessor system is abandoned and A<sub>1</sub> of the microprocessor system is connected with A<sub>0</sub> of the 8259A. Before going for an ALP, all the initialization command words (ICWs) and Operation Command Words (OCWs) must be decided. ICW<sub>1</sub> decides single level triggered, address interval of 4 as given below.



There is no slave hence the  $ICW_3$  is as given below:

$A_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$	$ICW_3 = 00H$
11	0	1	0	0	0	0	0	0	

Actually  $ICW_3$  is not at all needed, because in  $ICW_1$ , the 8259 A is set for single mode.

The  $ICW_4$  should be set as shown below:

$A_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$	$ICW_4 = 01H$
1	0	0	0	0	0	0	0	1	

For special fully nested mode masking

Non buffered mode

For 8086 system

Normal EOI

The  $OCW_1$  sets the mask of IR6 as below:

$A_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$
1	0	1	0	0	0	0	0	0

IR6 masked

$OCW_1 = 40H$

The  $OCW_2$  sets the modes and rotating priority as shown below:

$A_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$
0	1	1	1	0	0	1	0	0

Specific EOI command with rotating priority

Bottom priority level set at IR4

$OCW_2 = E4H$

The  $OCW_3$  sets the special mask mode and reads ISR and IRR using the following control commands:

For reading IRR -

For reading ISR

$A_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$
0	0	1	1	0	1	0	1	0

Special mask mode

No poll command

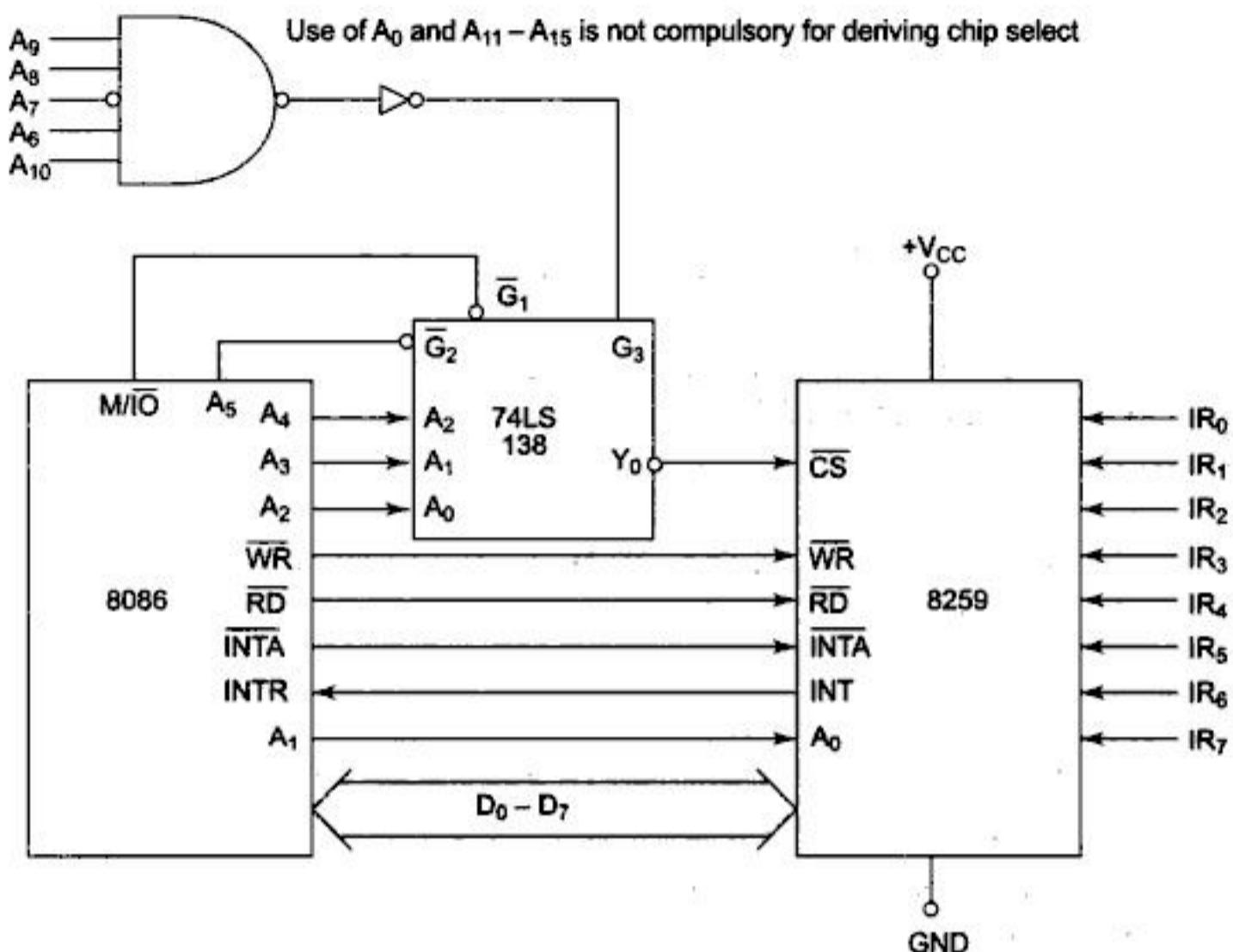
Read IRR

$OCW_3 = 6AH$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The interfacing circuit for Problem 6.3 has been shown below in Fig. 6.21.



**Fig. 6.21** Interfacing 8259A with 8086

#### Problem 6.4

Interface 3 ICS of 8259 PIC with 8086 system in such a way that one is master and rest two are slaves connected at  $IR_3$  and  $IR_6$  interrupt request level of the master. The 8259s are having vector address 60H, 70H and 80H. Write a program to initialize 8259 PIC so that  $IR_2$  and  $IR_7$  levels of master are masked. Initialize master in AEOI mode and automatic rotation mode in minimum mode of operation.

#### Solution

Address Decoding table for 8259 interfaced at even addresses.

**Table 6.3**

	Port	Hex. Address	$A_{15}$	$A_{14}$	$A_1$	$A_0$
Master	Port 0	C000H	1100	0000	0000	000 0
8259	Port 1	C002H	1100	0000	0000	001 0
Slave <sub>3</sub>	Port 0	0004H	1100	0000	0000	010 0
	Port 1	C006H	1100	0000	0000	011 0
Slave <sub>6</sub>	Port 0	C008H	1100	0000	0000	100 0
	Port 1	C00AH	1100	0000	0000	101 0

- \*  $A_1$  Pin of 8086 system is connected with  $A_0$  pin of 8259 PIC master as well as slave.
- \*  $A_3, A_2$  are used to generate chip select for three 8259 PIC.
- \*  $A_{15} - A_4$  used as shown in the decoder diagram
- \*  $A_0$  is used to generate chip select for even bank



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

For rotating priority in AEOI mode.

Program for this problem is presented below.

```

ASSUME CS : CODE, DS : DATA
STACK SEGMENT
TOP DW 100 DUP (?)      Stack segment of 200 by ks
STACK ENDS
CODE SEGMENT
START : MOV AX, STACK
        MOV SS, AX           Initialization of SS 3 SP
        LEA SP, TOP+200
        CLI
        MOV DX, 0C000H        Port 0 address of Master in DX
        MOV AL, 19H            ICWI of Master
        OUT DX, AL             Out to Port 0 of Master
        ADD DX, 02H            Port 1 address in Dx
        MOV AL, 60H            Vector address of Master for IR0 (ICW2)
        OUT DX, AL             ICW2 is out to Port 1
        MOV AL, 48H            ICW3 for Master
        OUT DX, AL             ICW3 is out to Port 1
        MOV AL, 13H            ICW4 is out to Port 1
        OUT DX, AL             OCWI for Master
        MOV AL, 84H            OCWI is Out to Port 1
        OUT DX, AL
        MOV AL, 80H            OCW2 for Master
        OUT DX, AL             OCW2 is out to Port 1
                           Now initialization of Slave 3.
        MOV DX, 0C004H        Port address of Slave 3 in DX
        MOV AL, 19H            ICWI
        OUT DX, AL
        ADD DX, 02H            Port 1 address of Slave 3 in Dx
        MOV AL, 70H            ICW2 for Slave 3
        OUT DX, AL
        MOV AL, 03H            ICW3 for Slave 3
        OUT DX, AL
        MOV AL, 01H            ICW4 for Slave 3
        OUT DX, AL             Now initialization of Slave 6
        MOV DX, 0C008H        Port 0 address of Slave 6 in Dx
        MOV AL, 19H            ICW1 for Slave 6
        OUT DX, AL
    
```

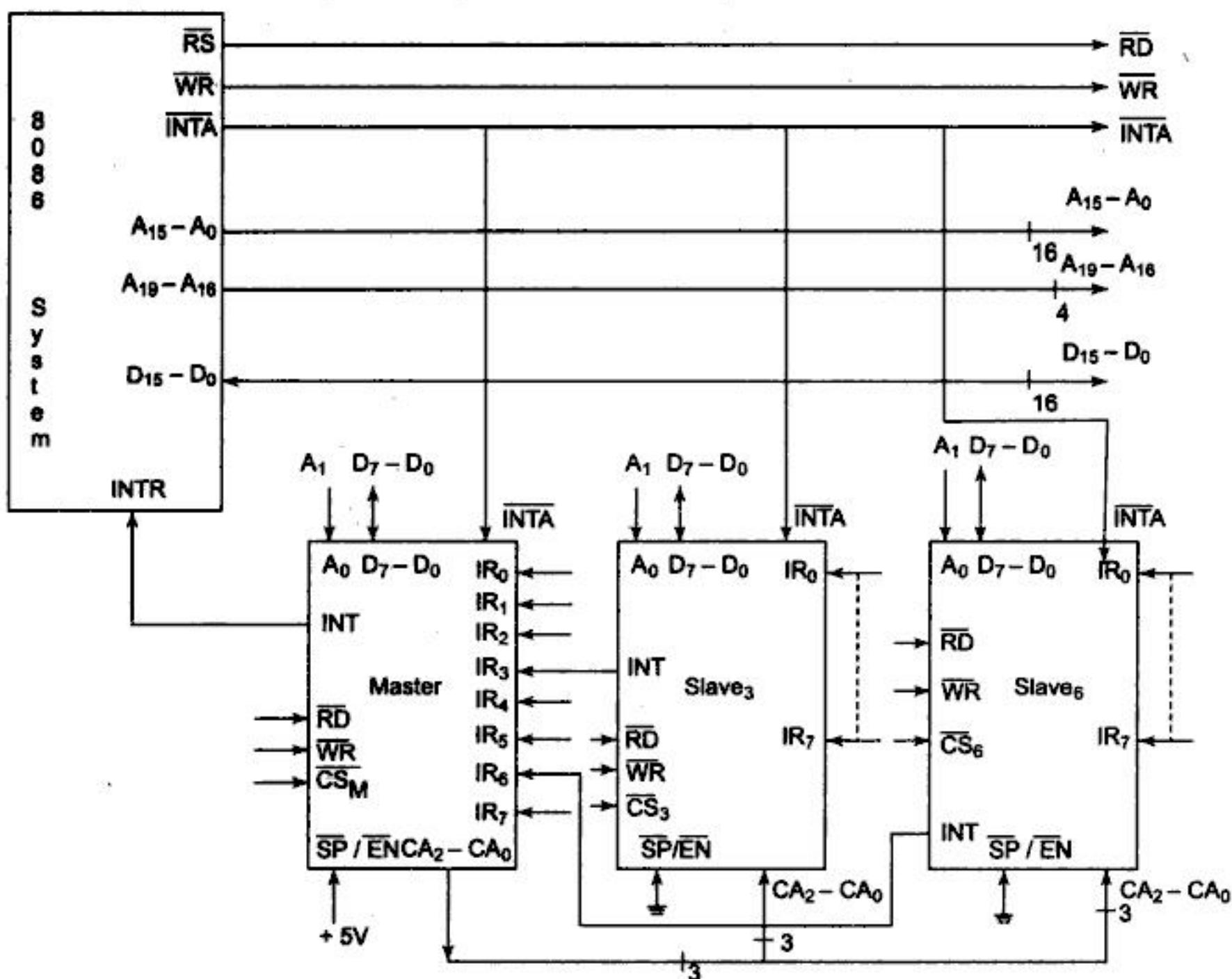
```

ADD DX, 02H          Port 1 address of Slave 6 in Dx
MOV AL, 80H          ICW2 for Slave 6
OUT DX, AL
MOV AL, 06H          ICW3 for Slave 6
OUT DX, AL
MOV AH, 01H          ICW4 for Slave 6
OUT DX, AL
MOV AH, 4CH
INT 21H
CODE ENDS
End start

```

**Program 6.6 ALP to Initialize Cascaded 8259A for Problem 6.4**

Interfacing circuit for this problem is presented below in Fig. 6.22.

**Fig. 6.22 Cascaded 8259 Interfacing for Problem 6.4**

### 6.3 THE KEYBOARD/DISPLAY CONTROLLER 8279

In Chapter 5, while studying 8255, we have explained the use of 8255 in interfacing keyboards and displays with 8086. The disadvantage of this method is that the processor has to refresh the display and check the status of the keyboard periodically using polling technique. Thus a considerable amount of CPU time is wasted, reducing the system operating speed and hence the throughput.

Intel's 8279 is a general purpose keyboard display controller that simultaneously drives the display of a system and interfaces a keyboard with the CPU, leaving it free for its routine task. The keyboard-display interface scans the keyboard to identify if any key has been pressed and sends the code of the pressed key to the CPU. It also transmits the data received from the CPU, to the display device. Both of these functions are performed by the controller in repetitive fashion without involving the CPU. The keyboard is interfaced either in the interrupt or the polled mode. In the interrupt mode, the processor is requested service only if any key is pressed, otherwise the CPU can proceed with its main task. In the polled mode, the CPU periodically reads an internal flag of 8279 to check for a key pressure. The keyboard section can interface an array of a maximum of 64 keys with the CPU. The keyboard entries (key codes) are debounced and stored in an 8-byte FIFO RAM, that is further accessed by the CPU to read the key codes. If more than eight characters are entered in the FIFO (i.e. more than eight keys are pressed), before any FIFO read operation, the overrun status is set. If a FIFO contains a valid key entry, the CPU is interrupted (in interrupt mode) or the CPU checks the status (in polling) to read the entry. Once the CPU reads a key entry, the FIFO is updated, i.e. the key entry is pushed out of the FIFO to generate space for new entries. The 8279 normally provides a maximum of sixteen 7-seg display interface with CPU. It contains a 16-byte display RAM that can be used either as an integrated block of  $16 \times 8$ -bits or two  $16 \times 4$ -bit blocks of RAM. The data entry to RAM block is controlled by CPU using the command words of the 8279.

#### 6.3.1 Architecture and Signal Descriptions of 8279

The keyboard display controller chip 8279 provides (a) a set of four scan lines and eight return lines for interfacing keyboards (b) a set of eight output lines for interfacing display. Figure 6.23 shows the functional block diagram of 8279 followed by its brief description.

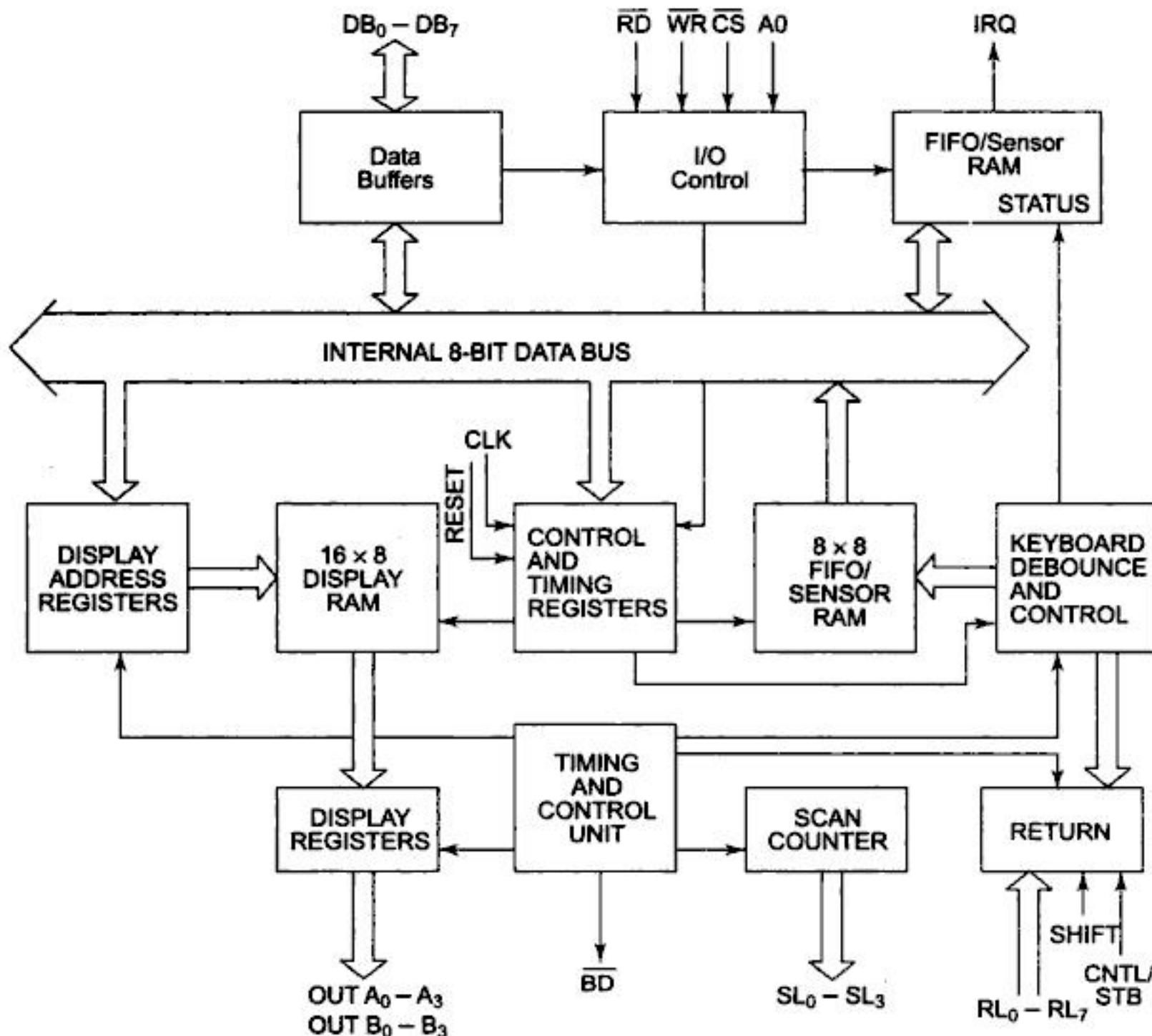
**I/O Control and Data Buffers** The I/O control section controls the flow of data to/from the 8279. The data buffers interface the external bus of the system with internal bus of 8279. The I/O section is enabled only if  $\overline{D}$  is low. The pins  $A_0$ ,  $\overline{RD}$  and  $\overline{WR}$  select the command, status or data read/write operations carried out by the CPU with 8279.

**Control and Timing Register and Timing Control** These registers store the keyboard and display modes and other operating conditions programmed by CPU. The registers are written with  $A_0 = 1$  and  $WR = 0$ . The timing and control unit controls the basic timings for the operation of the circuit. Scan counter divide down the operating frequency of 8279 to derive scan keyboard and scan display frequencies.

**Scan Counter** The scan counter has two modes to scan the key matrix and refresh the display. In the encoded mode, the counter provides a binary count that is to be externally decoded to provide the scan lines for keyboard and display (four externally decoded scan lines may drive up to 16 displays). In the decoded scan mode, the counter internally decodes the least significant 2 bits and provides a decoded

1 out of 4 scan on  $SL_0$ - $SL_3$  (four internally decoded scan lines may drive up to 4 displays). The keyboard and display both are in the same mode at a time.

**Return Buffers and Keyboard Debounce and Control** This section scans for a key closure row-wise. If it is detected, the keyboard debounce unit debounces the key entry (i.e. wait for 10 ms). After the debounce period, if the key continues to be detected. The code of the key is directly transferred to the sensor RAM along with SHIFT and CONTROL key status.



**Fig. 6.23** 8279 Internal Architecture

**FIFO/Sensor RAM and Status Logic** In keyboard or strobed input mode, this block acts as 8-byte first-in-first-out (FIFO) RAM. Each key code of the pressed key is entered in the order of the entry, and in the meantime, read by the CPU, till the RAM becomes empty. The status logic generates an interrupt request after each FIFO read operation till the FIFO is empty. In scanned sensor matrix mode, this unit acts as sensor RAM. Each row of the sensor RAM is loaded with the status of the corresponding row of sensors in the matrix. If a sensor changes its state, the IRQ line goes high to interrupt the CPU.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



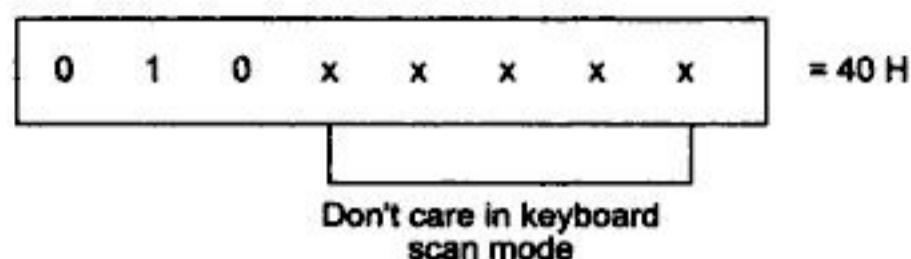
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



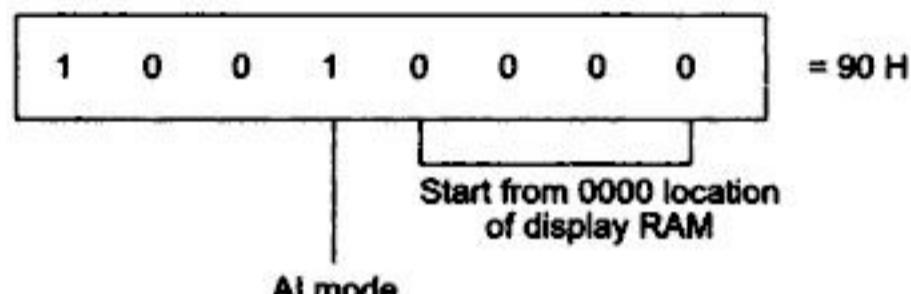
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Write Display RAM** This command enables the programmer to write the addressed display locations of the RAM as presented below.



AI mode

Program 6.7 gives the ALP required to initialise the 8279 as required.

```

ASSUME CS : CODE
CODE SEGMENT
START: MOV AL, 1AH      ; Set 8279 in Encoded scan,
        OUT 82H, AL      ; N key rollover, 16 display, Right entry mode.
        MOV AL, 34H      ; Set clock prescalar to
        OUT 82H, AL      ; 100 kHz
        MOV AL, 0D3H      ; Clear display ram
        OUT 82H, AL      ; command
        MOV AL, 40H      ; Read FIFO command
        OUT 82H, AL      ; for checking display RAM
WAIT:   IN AL, 82H      ; Wait for clearing of
        AND AL, 80H      ; Display RAM by reading
        CMP AL, 80H      ; FIFO Du bit of the status word i.e.
        JNZ WAIT         ; If DU bit is not set wait, else proceed
        MOV AL, 40H      ; Read FIFO command
        OUT 82H, AL      ; for checking key closure
        IN AL, 82H      ; Read FIFO status
        AND AL, 07H      ; Mask all bits except the
        CMP AL, 00        ; number of characters bits
        JNZ KEYCODE     ; If any key is pressed, take
WRAM:   MOV AL, 90H      ; required action, otherwise
        OUT 82H, AL      ; proceed to write display
        MOV AL, 55H      ; RAM by using write display

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The data buffer interfaces the internal bus of the circuit with the system bus. The read write control logic controls the operation of the peripheral depending upon the operations initiated by the CPU. This unit also selects one of the two internal addresses those are control address and data address at the behest of the C/D signal. The modem control unit handles the modem handshake signals to coordinate the communication between the modem and the USART. The transmit control unit transmits the data byte received by the data buffer from the CPU for further serial communication. This decides the transmission rate which is controlled by the TXC input frequency. This unit also derives two transmitter status signals namely TXRDY and TXEMPTY. These may be used by the CPU for handshaking. The transmit buffer is a parallel to serial converter that receives a parallel byte for conversion into a serial signal and further transmission onto the communication channel. The receive control unit decides the receiver

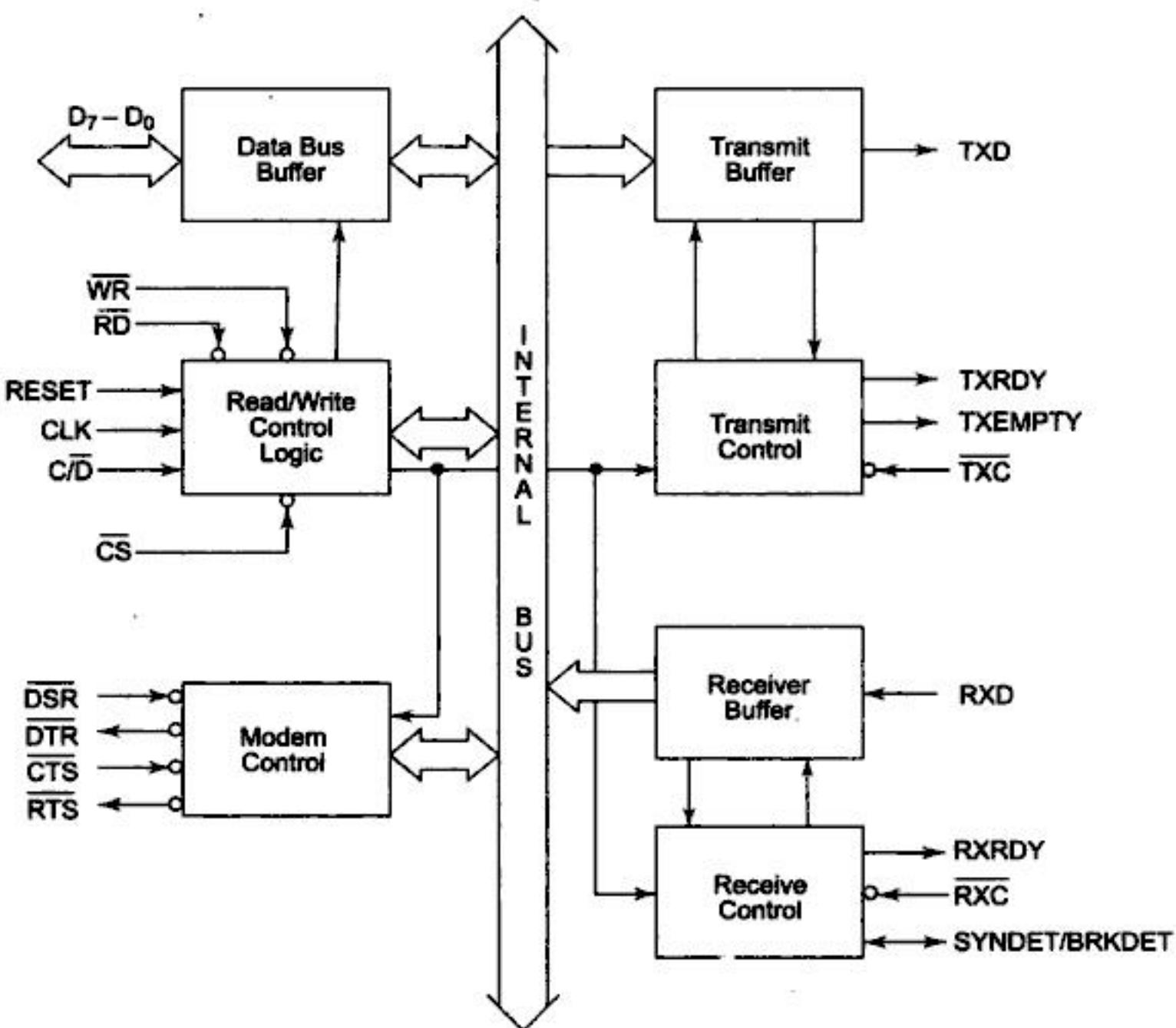


Fig. 6.26 8251A Internal Architecture

frequency as controlled by the RXC input frequency. This unit generates a receiver ready (RXRDY) signal that may be used by the CPU for handshaking. This unit also detects a break in the data string while the 8251 is in asynchronous mode. In synchronous mode, the 8251 detects SYNC characters using SYNDET/BD pin.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

edge until it matches. If 8251A is programmed for two SYNC characters, the subsequent received character is also checked. When both the characters match, the hunting stops. The SYNDET pin is set high and is reset automatically by a status read operation. If a parity bit is programmed, the SYNDET signal will not go as high as the middle of parity bit, or till the middle of the last data bit.

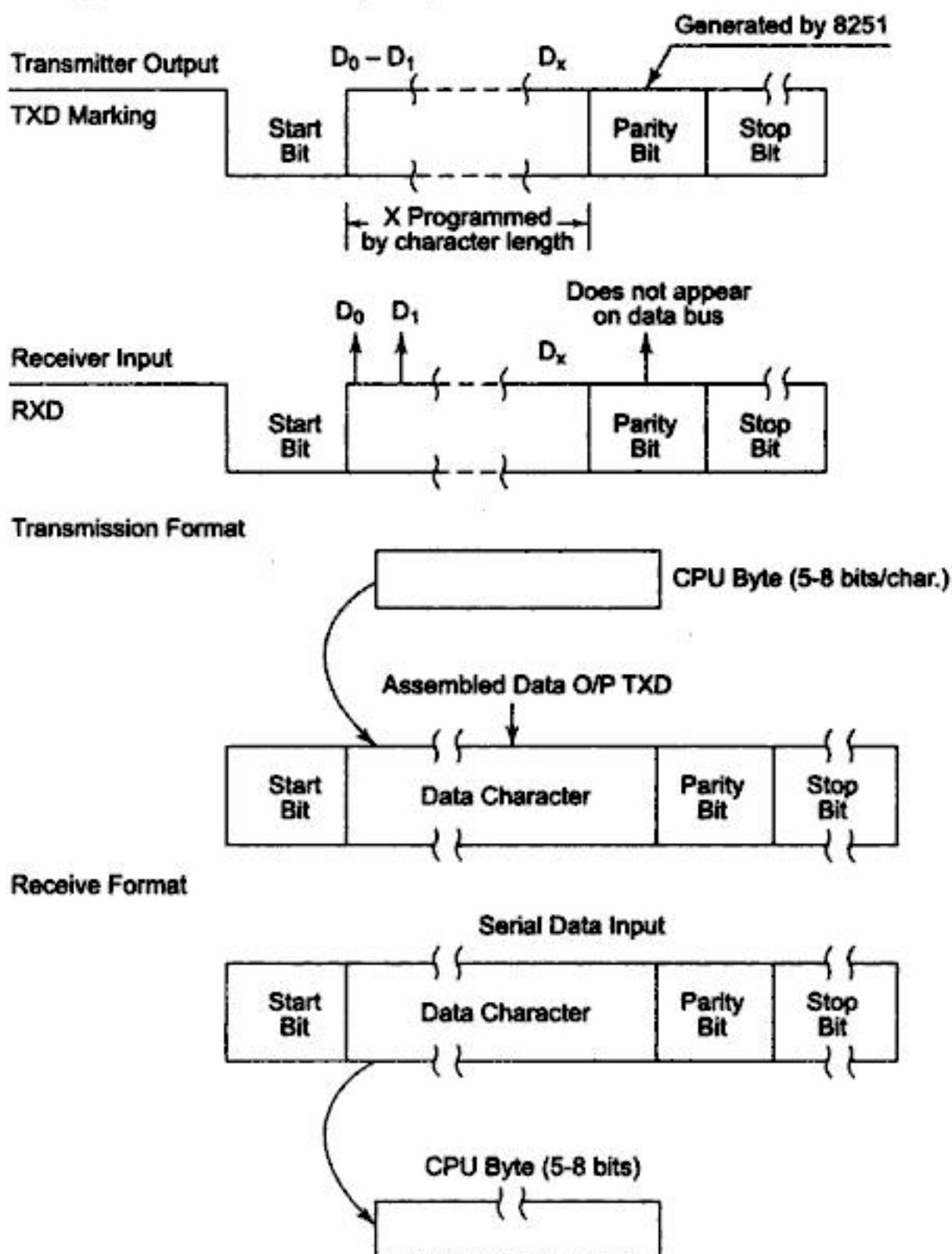
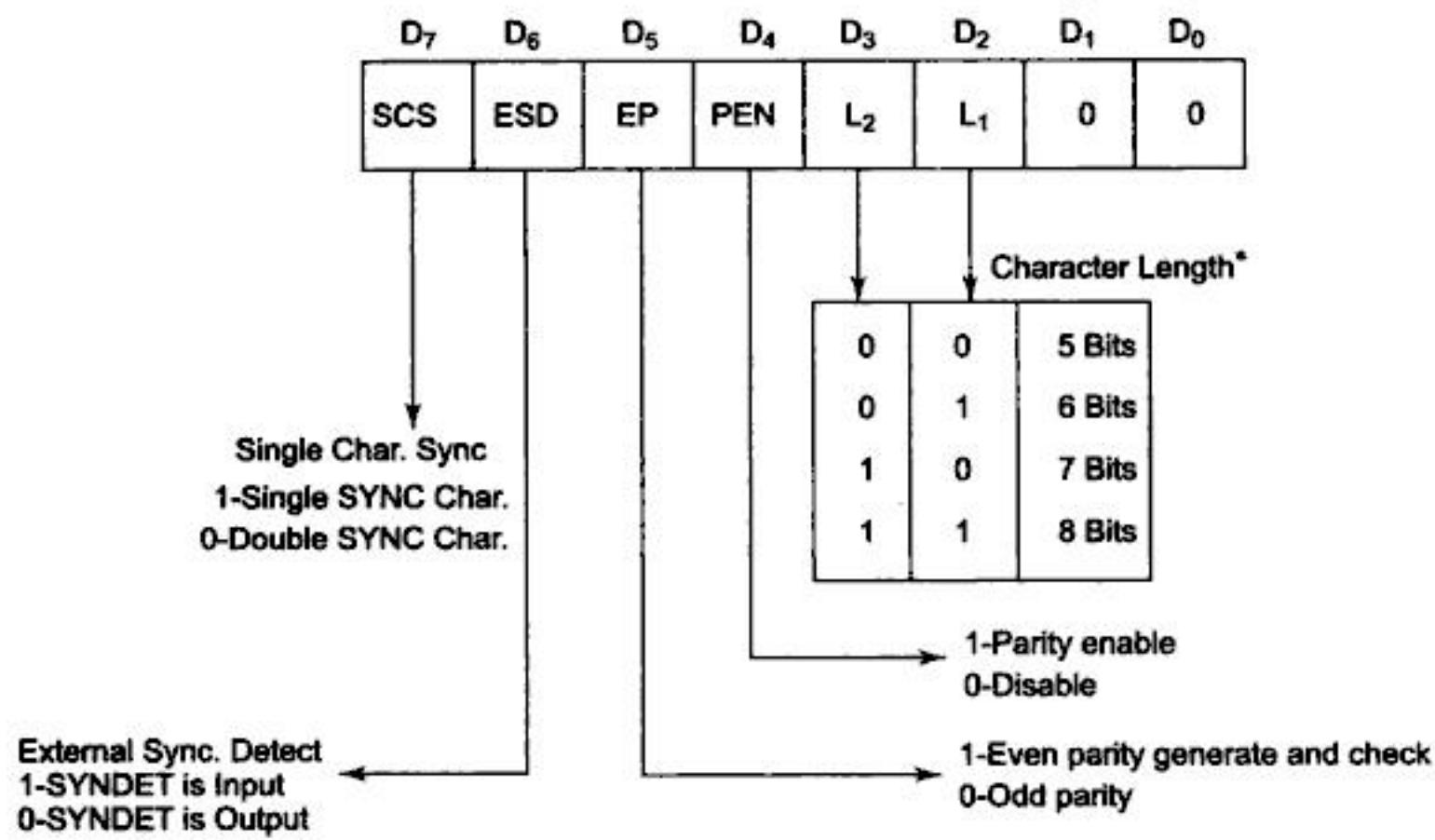


Fig. 6.29 Asynchronous Mode Transmit and Receive Formats

In the external SYNC mode, synchronization is achieved by applying a high level on the SYNDET input pin, that forces 8251A out of HUNT mode. The high level can be removed after one RXC cycle. An ENTER HUNT command has no meaning in asynchronous mode. The parity and overrun error both are checked in the same way as in asynchronous mode. Figure 6.32 shows synchronous mode transmit and receive data formats.

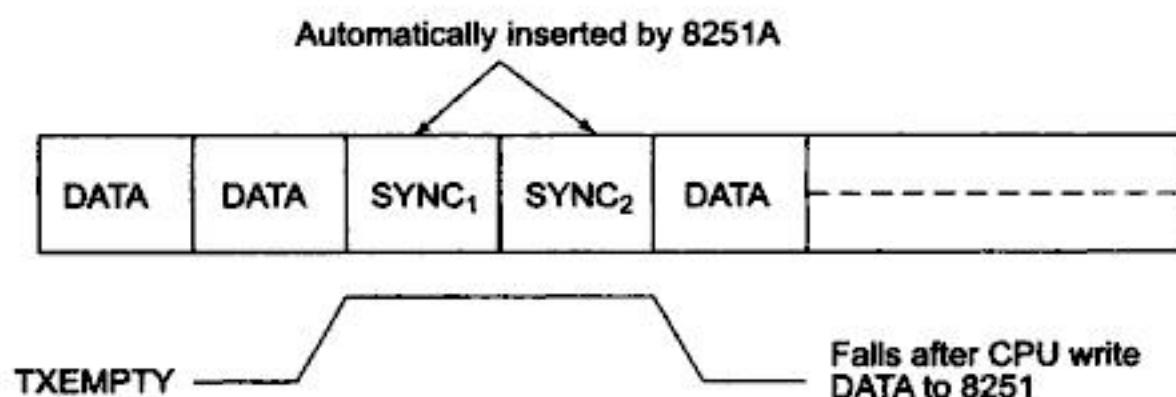
**Command Instruction Definition** The command instruction controls the actual operations of the selected format like enable transmit/receive, error reset and modem control. Once the mode instruction

has been written into 8251A and the SYNC characters are inserted internally by 8251A, all further control words written with  $C/D = 1$  will load a command instruction. A reset operation returns 8251A back to mode instruction format. The command instruction format is shown in Fig. 6.33, with its bit definitions.



\* If the character size less than 8-bits, the remaining bits are set to '0'.

**Fig. 6.30 Synchronous Mode Instruction Format**



**Fig. 6.31 TXEMPTY Signal and SYNC Characters**

**Status Read Definition** This definition is used by the CPU to read the status of the active 8251A to confirm if any error condition or other conditions like the requirement of processor service has been detected, during the operation.

A read command is issued by processor with  $C/D = 1$  to accomplish this function. Some of the bits in the definition have the same significances as those of the pins of 8251A. These are used to interface the 8251A in a polled configuration, besides the interrupt controlled mode. The pin TXRDY is an exception. The status 'read format' is shown in Fig. 6.34, with its bit definitions.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

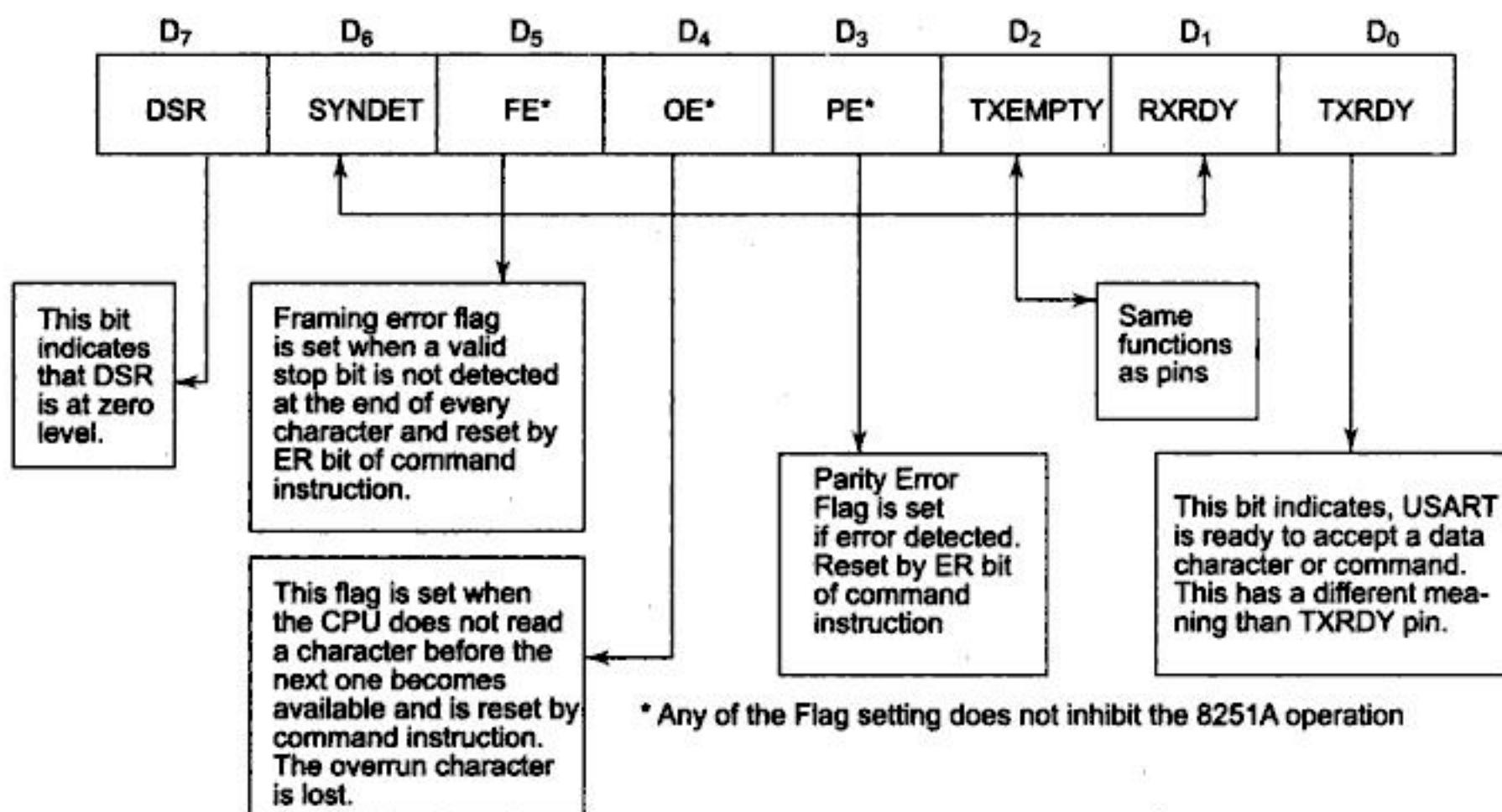


Fig. 6.34 Status Read Instruction Format

#### 6.4.4 Interfacing and Programming 8251 with 8086

The following problem explains the interfacing and programming of 8251A in an 8086 system.

##### Problem 6.6

Design the hardware interface circuit for interfacing 8251 with 8086. Set the 8251A in asynchronous mode as a transmitter and receiver with even parity enabled, 2 stop bits, 8-bit character length, frequency 160 kHz and baud rate 10 K.

- (a) Write an ALP to transmit 100 bytes of data string starting at location 2000:5000H.
- (b) Write an ALP receive 100 bytes of data string and store it at 3000:4000H.

##### Solution

The interfacing connections of 8251A with 8086 are shown in Fig. 6.35.

Asynchronous mode control word for Problem 6.6 (a)

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	= 0FE H
1	1	1	1	1	1	1	0	

2 stop              Even parity              8-bit              CLK scaled  
bits              enabled              format              by 16

- (a) ALP to initialize 8251 and transmit 100 bytes of data

```

ASSUME CS : CODE
CODE SEGMENT
START: MOV AX,2000H    ;
       MOV DS,AX      ; DS points to byte string segment

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

READY:    IN AL,0FEH      ; Check RXRDY. If the
          AND 02H        ; receiver is not ready,
          JZ READY       ; wait
          IN AL,0FCH      ; If it is ready,
          MOV [SI],AL      ; receive the character
          INC SI          ; Increment pointer to next byte
          DEC CL          ; Decrement counter
          JNZ NXTBT      ; Repeat, if CL is not zero
          MOV AH,4CH      ; If CL is 0, return to DOS
          INT 21H

CODE      ENDS

END START

```

Program 6.9 ALP to Receive 100 Bytes

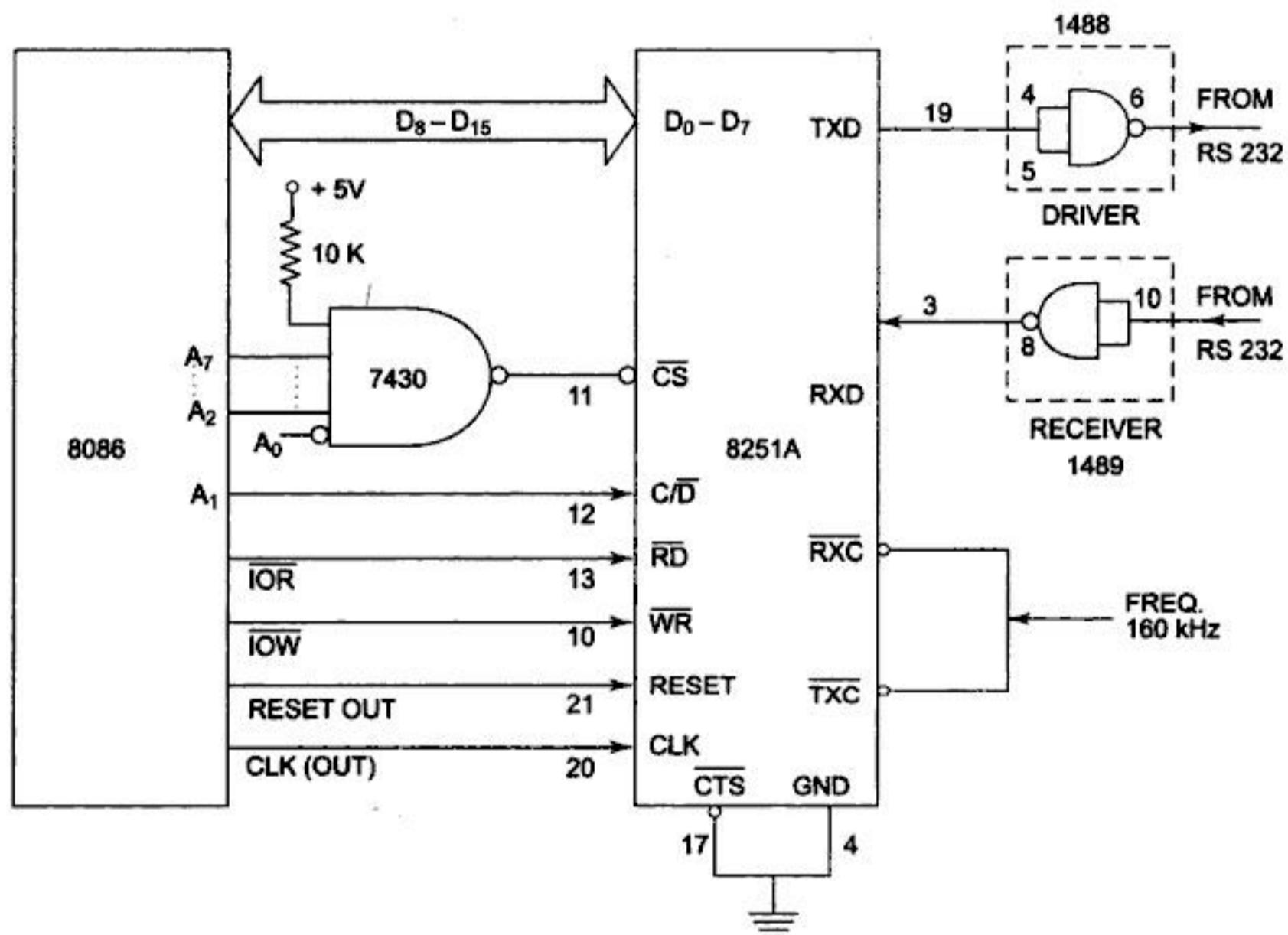


Fig. 6.35 Interfacing of 8251A with 8086

**SUMMARY**

In this chapter, we have presented a detailed account of the functioning of some of the important Intel peripherals. With the recent advances in the field, the Intel family of the peripherals, has been continuously increasing. Those, which are frequently used in the industrial and general systems are discussed in this chapter, in significant details. This chapter has been started with the discussion on the programmable timer 8253. The necessary functional details of 8253 have been discussed along with an interfacing example and supporting programs. Further, the peripherals like programmable interrupt controller 8259A, programmable keyboard display controller 8279A, programmable communication interface 8251A have been discussed along with their architectures, signal descriptions, interfacing and programming examples.

Thus this chapter has provided an insight into the operations, programming and interfacing of the dedicated peripherals.

**Exercises**

- 6.1 Draw and discuss the internal architecture of 8253.
- 6.2 Draw and discuss the different modes of operation of 8253.
- 6.3 Explain the significances of different bits of the control word register format of 8253.
- 6.4 Design a real time clock using 8253 interfaced with 8086. The CLK input to the 8253 is of 1.5 MHz frequency. Assume suitable addresses for 8253. Further, display the time using a 6-digit 7-segment multiplexed display unit which is interfaced with the 8086 using 8255. The 7-segment data port address of 8255 is 00 and the digit select port address of 8255 is 01. Draw the hardware schematic and write a program.
- 6.5 Design an 8086 microprocessor based stop-watch using 8253 and 8255. The stop-watch counts up to 100 seconds in the steps of 10 ms and displays the time on a four-digit 7-segment multiplexed display. The CLK input frequency to 8253 is 2.4 MHz. Draw the required hardware scheme and write the required ALP. Select suitable addresses for 8253 and 8255.
- 6.6 A flow transducer, which generates number of TTL compatible pulses proportional to the volume of the liquid passed through it, is available. It generates a pulse, if 5 ml volume of the liquid passes through it. Design an 8086 based system using 8253 for measuring up to 1000 litres of the liquid at a precision of 10 ml. Assume the required addresses suitably.
- 6.7 Draw and discuss the internal architecture of 8259A.
- 6.8 Explain the functions of the following pins of 8259A.
  - (i)  $CAS_0-CAS_2$
  - (ii)  $\overline{SP}/\overline{EN}$
- 6.9 Describe an interrupt request response of an 8086 system.
- 6.10 What is the difference between 8259 and 8259A?
- 6.11 Explain the initialisation sequence of 8259A.
- 6.12 How will you provide more than eight interrupt input lines to an 8086 based system?  
Design an interrupt system which provides twenty nine interrupt inputs to the 8086 system.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- 6.29 Interface a 26-keys keyboard with 8279. The keys represent the alphabets 'a' to 'z'. Write an 8086 ALP to find out the ASCII equivalent of the alphabet corresponding to the pressed key. The 8086 system runs at 6 MHz while the 8279 should work at 200 kHz. Will the internal prescalar reduce 6 MHz to 200 kHz? If any external prescalar is required, design it with minimum hardware.
- 6.30 Draw and discuss internal architecture of USART 8251.
- 6.31 Explain the following signal descriptions of 8251.

(i) C/D

(ii) TXC

(iii) TXD

(iv) RXC

(v) RXD

(vi) RXRDY

(vii) TXRDY

(viii) DSR

(ix) DTR

(x) RTS

(xi) CTS

(xii) TXEMPTY

(xiii) SYNDET/BD

- 6.32 Explain the mode instruction control word format of 8251.
- 6.33 Draw and discuss the asynchronous mode transmitter and receiver data formats of 8251.
- 6.34 Draw and discuss the status word format of 8251.
- 6.35 Draw and discuss the synchronous mode transmit and receive data formats of 8251.
- 6.36 Interface 8251 with 8086 at an address 80H. Initialise it in asynchronous transmit mode, with 7-bits character size, baud factor 16, one start bit, one stop bit, even parity enabled. Further transmit a message 'HAPPY NEW YEAR' in ASCII coded form to a modem.
- 6.37 Write a program to initialise 8251 in synchronous mode with even parity, single SYNCH character, 7-bit data character. Then receive FFH bytes of data from a remote terminal and store it in the memory at address 5000H:2000H.

# DMA, Floppy Disk and CRT Controllers

## INTRODUCTION

In the previous chapter, we studied some of the dedicated peripherals and their interfacing techniques with 8086. In this chapter, we will further discuss a few advanced peripherals and their interfacing techniques with 8086. In the applications where the CPU is to transfer bulk data, it may be a waste of time to transfer the data from source to destination using program controlled data transfer or interrupt driven data transfer. The alternate way of transferring the bulk data is the Direct Memory Access (DMA) technique in which the data is transferred under the control of a DMA controller, after it is properly initialised by the CPU. A DMA controller is designed to complete the bulk data transfer task much faster than the CPU. One such application which involves bulk data transfer is the storage of programs or data into secondary memory.

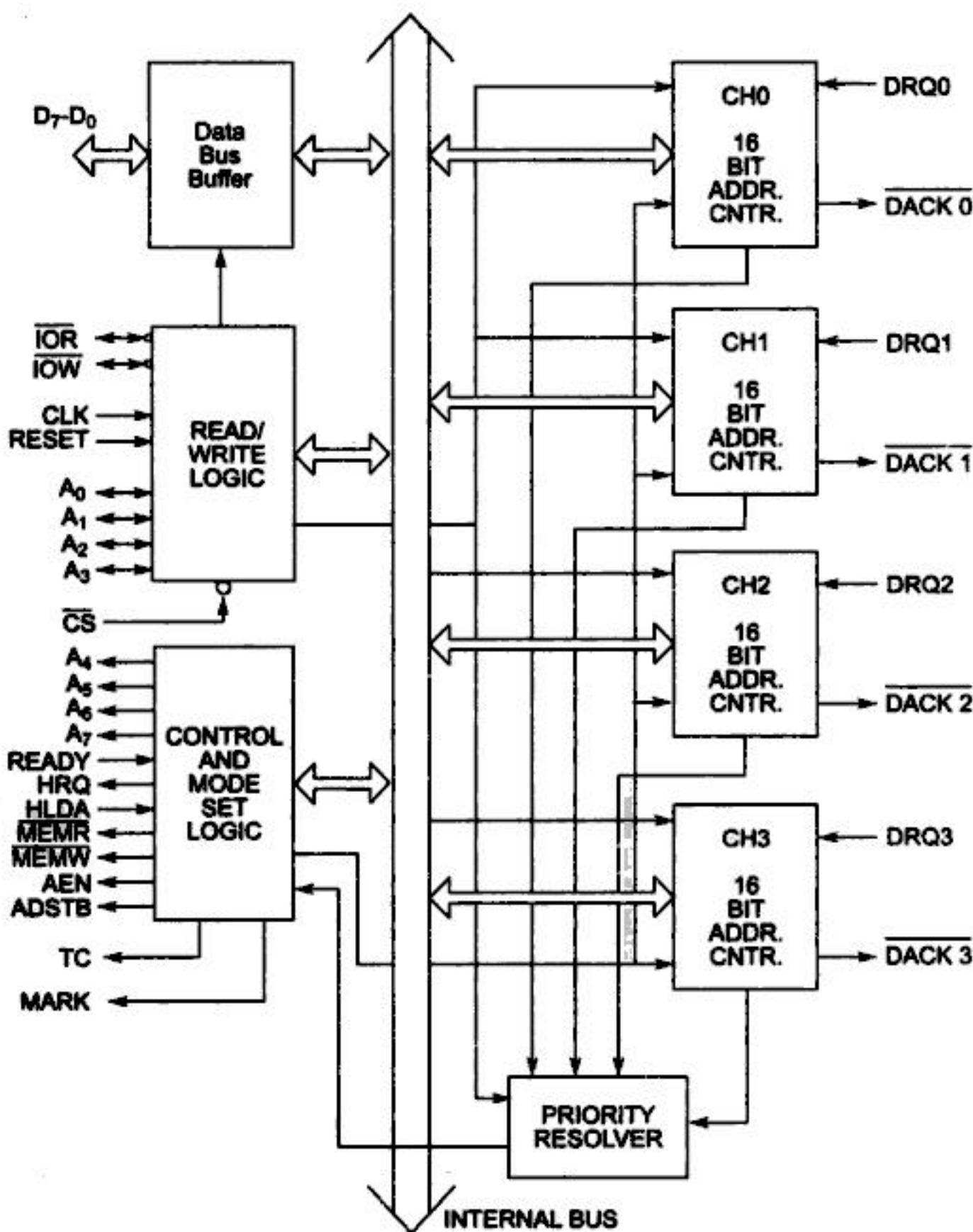
The floppy disk is one of the most popular forms of secondary memories. A Floppy Disk Controller (FDC) coordinates the complicated task of interfacing the floppy disk drive mechanism with the CPU, storing the parallel data onto the magnetic disk media in a serial bit string form and also reading the serially stored data and converting it into the parallel form. This data transfer between the CPU and the FDC may be controlled using a DMA controller.

A CRT controller derives all the control and timing signals required for controlling and interfacing a CRT with the CPU. A DMA controller may also be used to transfer data from the system memory to a video RAM of a CRT display.

In short, this chapter elaborates DMA controllers and other peripherals which may require DMA for their operation, viz. floppy disk controller and CRT controllers.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



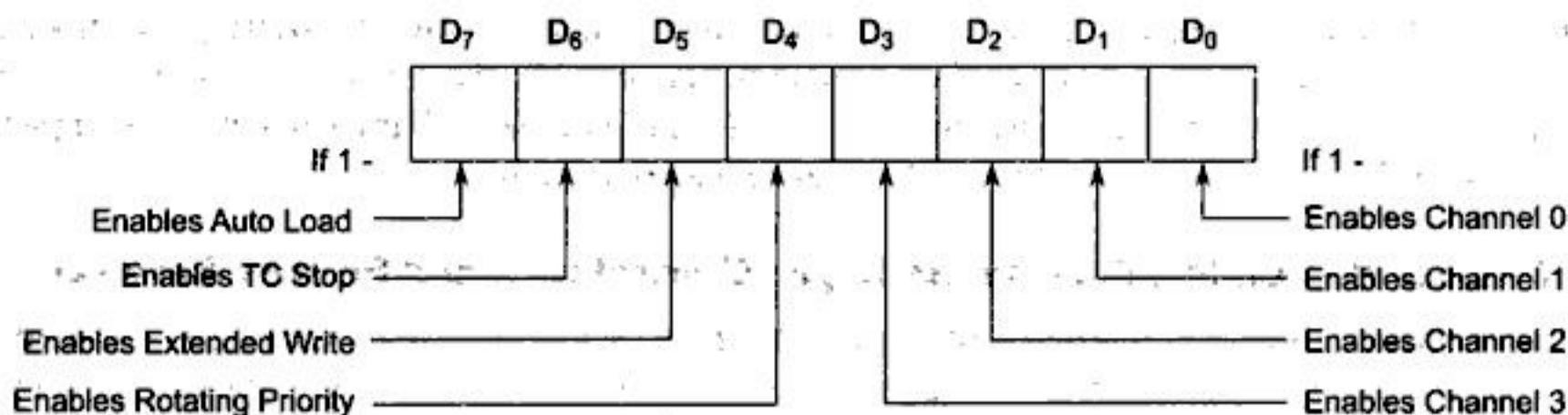
**Fig. 7.1 Internal Architecture of 8257**

The bits 14 and 15 of this register indicate the type of the DMA operation (transfer). If the device wants to write data into the memory, the DMA operation is called DMA write operation. Bit 14 of the register in this case will be set to one and bit 15 will be set to zero. Table 7.2 gives details of DMA operation selection and the corresponding bit configuration of the bits 14 and 15 of the TC register.

**Mode Set Register** The mode set register is used for programming the 8257 as per the requirements of the system. The function of the mode set register is to enable the DMA channels individually and also to set the various modes of operation. A DMA channel should not be enabled till the DMA address register and the terminal count register contain valid information, otherwise, an unwanted DMA request may initiate a DMA cycle, probably destroying the valid memory data.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



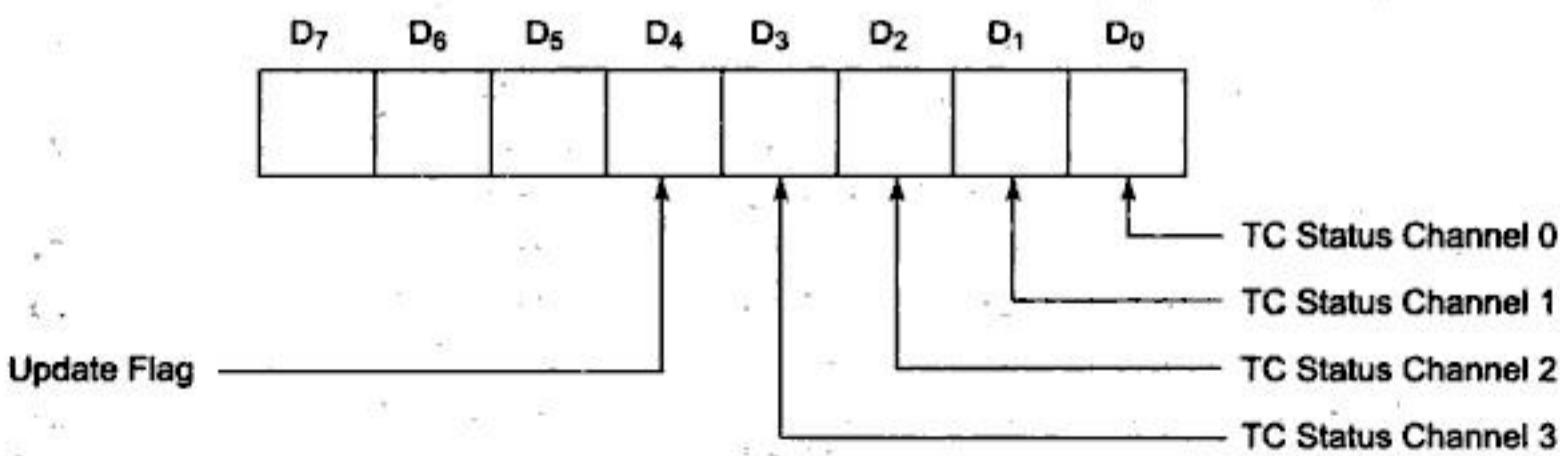
**Fig. 7.2 Bit Definitions of the Mode Set Register**

If the TC STOP bit is set, the selected channel is disabled after the *terminal count* condition is reached, and it further prevents any DMA cycle on the channel. To enable the channel again, this bit must be reprogrammed. If the TC STOP bit is programmed to be zero, the channel is not disabled, even after the count reaches zero and further requests are allowed on the same channel.

The auto load bit, if set, enables channel 2 for the repeat block chaining operations, without immediate software intervention between the two successive blocks. The channel 2 registers are used as usual, while the channel 3 registers are used to store the block reinitialisation parameters, i.e. the DMA starting address and the terminal count. After the first block is transferred using DMA, the channel 2 registers are reloaded with the corresponding channel 3 registers for the next block transfer, if the *Update* flag is set.

The extended write bit, if set to '1', extends the duration of **MEMW** and **IOW** signals by activating them earlier. This is useful in interfacing the peripherals with different access times. If the peripheral is not accessed within the stipulated time, it is expected to give the 'NOT READY' indication to 8257, to request it to add one or more wait states in the DMA cycle. The mode set register can only be written into.

**Status Register** The status register of 8257 is shown in Fig. 7.3. The lower order 4-bits of this register contain the terminal count status for the four individual channels. If any of these bits is set, it indicates that the specific channel has reached the terminal count condition. These bits remain set till either the status is read by the CPU or the 8257 is reset. The update flag is not affected by the read operation. This flag can only be cleared by resetting 8257 or by resetting the auto load bit of the mode set register. If the update flag is set, the contents of the channel 3 registers are reloaded to the corresponding registers of



**Fig. 7.3 Bit Definitions of Status Register of 8257**



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

to inhibit the non-DMA devices from responding during DMA operations. This also may be used to transfer the higher byte of the generated address over the data bus. If the 8257 is I/O mapped, this should be used to disable the other I/O devices, when the DMA controller address is on the address bus.

**TC** Terminal count output indicates to the currently selected peripheral that the present DMA cycle is the last for the previously programmed data block. If the TC STOP bit in the mode set register is set, the selected channel will be disabled at the end of the DMA cycle. The TC pin is activated when the 14-bit content of the terminal count register of the selected channel becomes equal to zero. The lower order 14 bits of the terminal count register are to be programmed with a 14-bit equivalent of  $(n-1)$ , if  $n$  is the desired number of DMA cycles.

**MARK** The modulo 128 mark output indicates to the selected peripheral that the current DMA cycle is the 128th cycle since the previous MARK output. The mark will be activated after each 128 cycles or integral multiples of it from the beginning of the data block (the first DMA cycle), if the total number of the required DMA cycles ( $n$ ) is completely divisible by 128.

**V<sub>cc</sub>** This is a +5V supply pin required for operation of the circuit.

**GND** This is a return line for the supply (ground pin of the IC).

## 7.2 DMA TRANSFERS AND OPERATIONS

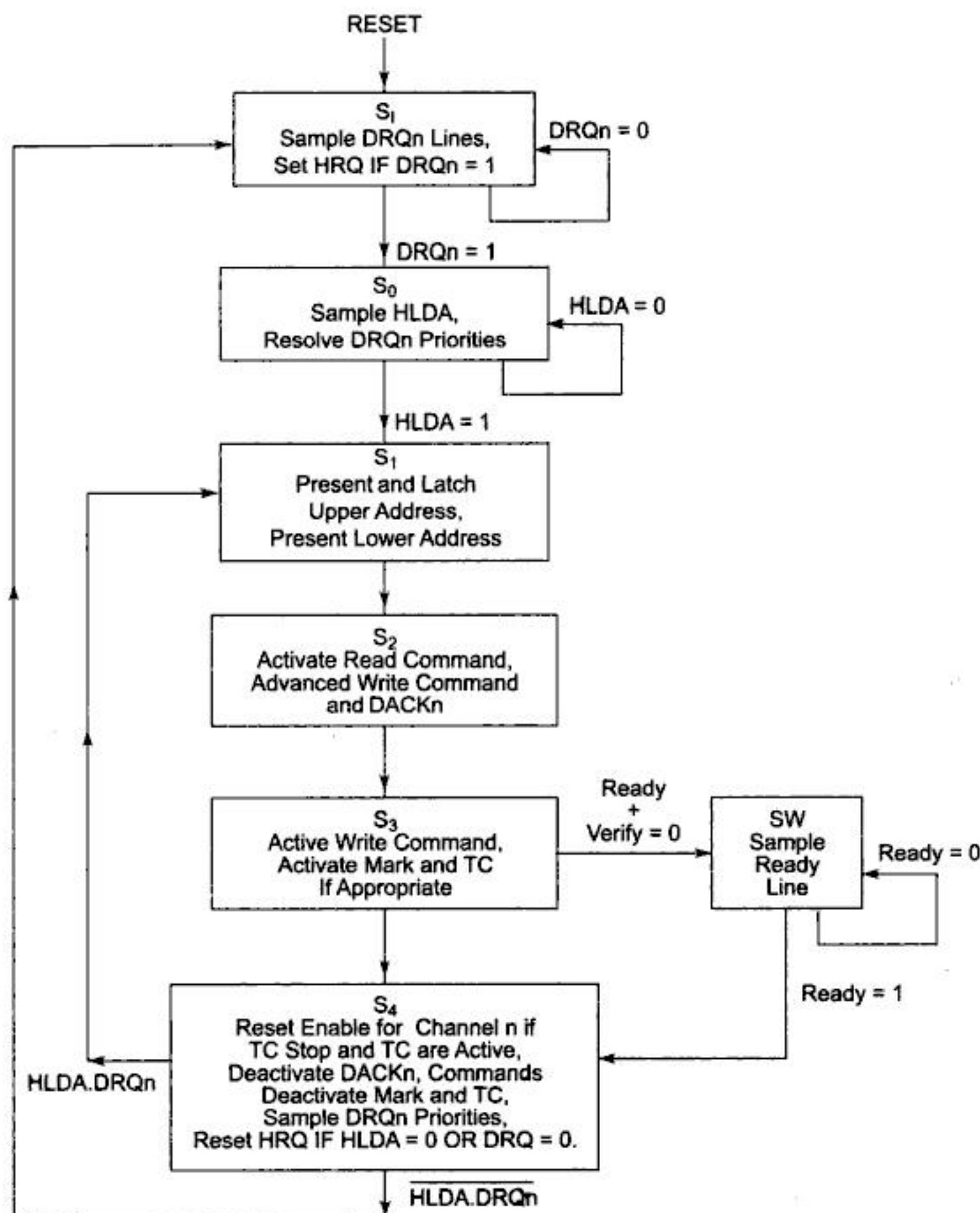
The 8257 is able to accomplish three types of operations, viz. verify DMA operation, write operation and read operation. The complete operational sequence of 8257 is described using a state diagram in Fig. 7.5 for a single channel.

A single byte transfer using 8257 may be requested by an I/O device using any one of the 8257 DRQ inputs. In response, the 8257 sends HRQ signal to the CPU at its HLD input and waits for acknowledgement at the HLDA input. If the HLDA signal is received by the DMA controller, it indicates that the bus is available for the transfer. The DACK line of the used channel is pulled down by the DMA controller to indicate the I/O device that its request for the DMA transfer has been honoured by the CPU. The DMA controller generates the read and write commands to transfer the byte from/to the I/O device. The DACK line is pulled down to indicate the DMA controller that the transfer, as requested by the device, is over. The HRQ line is lowered by the DMA controller to indicate the CPU that it may regain the control of the bus. The DRQ must be high until acknowledged and must go low before S<sub>4</sub> state of the DMA operation state diagram to avoid another unwanted transfer.

If more than one channel requests service simultaneously, the transfer will occur as a *burst transfer*. This will be discussed further in case of 8237. No overhead is required in switching from one channel to another. In each S<sub>4</sub>, the DRQ lines are sampled and the highest priority request is recognized during the next transfer. Once the higher priority transfer is over; the lower priority transfer requests may be served, provided their DRQ lines are still active. The HRQ line is maintained active till all the DRQ lines go low.

The burst or continuous transfer, described above may be interrupted by an external device by pulling down the HLDA line. After each transfer, the 8257 checks the HLDA line. If it is found active, it completes the current transfer and releases the HRQ line (i.e. sends it low) and returns to its idle state. If the DRQ line is still active, the 8257 will again activate HRQ and proceed as already described. The

8257 uses four clock cycles to complete a transfer. The 8257 has a READY input to interface it with low speed devices. The READY pin status is checked in  $S_3$  of the state diagram. If it is low, the 8257 enters a wait state. The status is sampled in every state till it goes high. Once the READY pin goes high, the 8257 continues from state  $S_4$  to complete the transfer. The 8257 can be interfaced as a memory mapped device or an I/O mapped device. If it is connected as memory mapped device, proper care must be taken while programming Rd/A<sub>15</sub> and Wr/A<sub>14</sub> bits in the terminal count register.



DRQn refers to any DRQ line of an enabled DMA channel

**Fig. 7.5 DMA Operation State Diagram**

### 7.2.1 Priorities of the DMA Requests

The 8257 can be programmed to select any of the two priority schemes using the command register. The first is the *fixed priority scheme*, while the second is the *rotating priority scheme*. In the fixed priority scheme, each device connected to a channel is assigned a fixed priority. In this scheme, the  $DREQ_3$  has the lowest priority followed by  $DRQ_2$  and  $DRQ_1$ .  $DRQ_0$  has the highest priority. In the rotating priority mode, the priorities assigned to the channels are not fixed. At any point of time, suppose  $DRQ_0$  has highest priority and  $DRQ_3$  the lowest, then after the device at channel 0 gets the service, its priority goes down and the channel 0 becomes the lowest priority channel. Channel 1 now becomes the highest priority channel, and remains the highest priority channel till it gets the service. Once channel 1 is served, it becomes the lowest priority channel and the channel 2 now becomes the highest priority channel. If you select the rotating priority, in a single chip DMA system any device requesting the service is guaranteed to be recognized after no more than three higher priority requests, thus avoiding dominance of any one channel. The priority allotment in the rotating priority mode is as shown in Fig. 7.6.

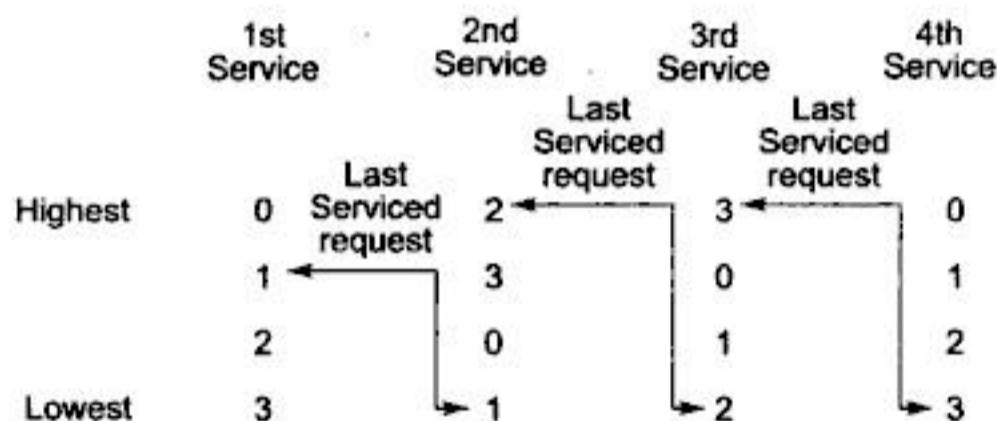


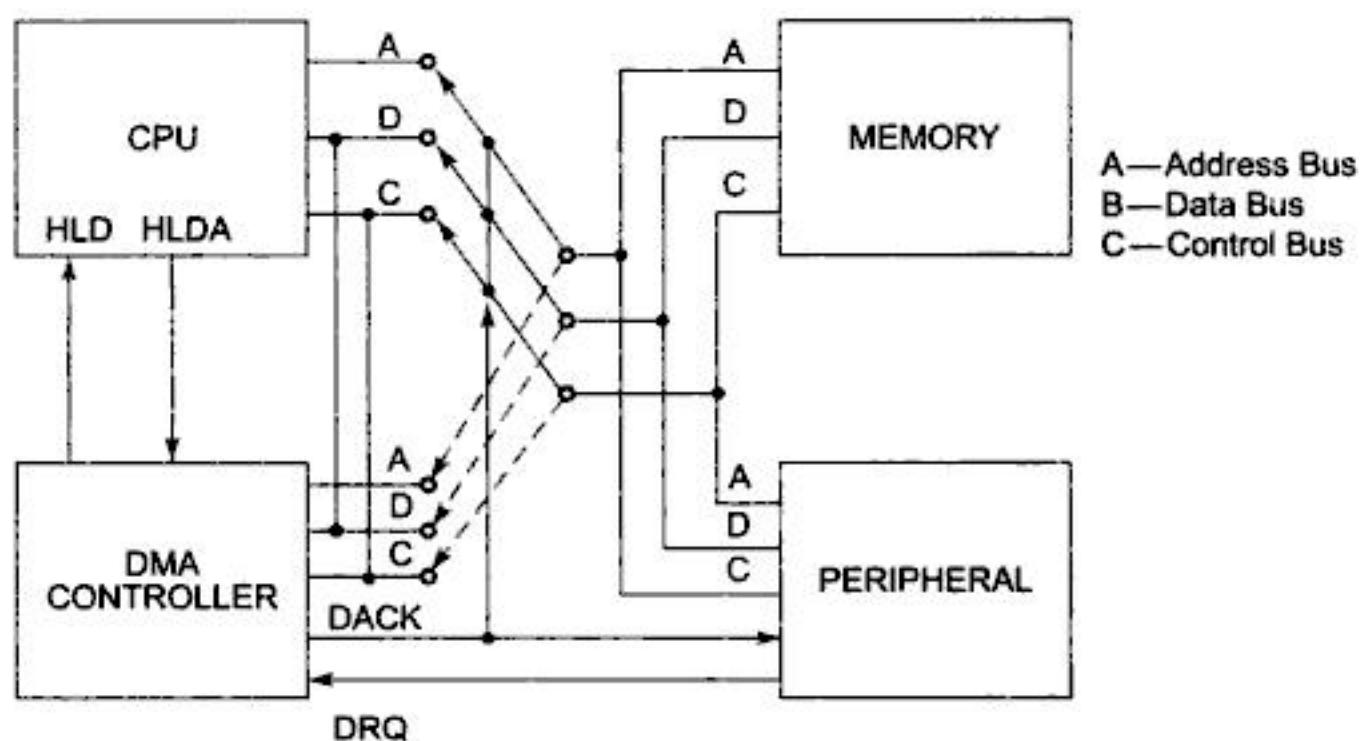
Fig. 7.6 Priority Allotment in Rotating Priority Mode

### 7.2.2 Programming and Reading the 8257 Registers

The selected register may be read or written depending upon the instruction executed by the CPU but the mode set register can only be written in, while the status register can only be read. The 16-bit register pair of each channel is read or written in two successive read or write operations. The Least Significant Byte (LSB) and the Most Significant Byte (MSB) of each register for a specific channel has the same address, but they are differentiated by an internal First/Last (F/L) flip-flop. If the F/L flip-flop is 0, it indicates the first operation, i.e. the LSB is to be read or written, otherwise, it is the last operation, i.e. the MSB is to be read or written. The F/L flip-flop can be cleared by resetting 8257. Thus the first operation after RESET will always be a LSB operation and the successive one for the same register address will be a MSB operation. The least significant three address bits  $A_0$ - $A_2$  indicate the specific register for a specific channel. The  $A_3$  address line is used to differentiate between all the channel registers and the common registers, i.e. mode set and status registers. The higher order address lines  $A_4$ - $A_{15}$  may be used to derive the chip select signal  $\overline{CS}$  of 8257. All the accesses to any of the terminal count registers and DMA address registers must be in pairs, i.e. the LSB accesses must be followed by the MSB accesses. In verify transfer mode, no actual data transfer takes place. In this mode, the 8257 acts in the same way as read or write transfer to generate addresses, but no control lines are activated.

### 7.2.3 Interfacing 8257 with 8086

Once a DMA controller is initialised by a CPU properly, it is ready to take control of the system bus on a DMA request, either from a peripheral or itself (in case of memory-to-memory transfers). The DMA controller sends a HOLD request to the CPU and waits for the CPU to assert the HLDA signal. The CPU relinquishes the control of the bus before asserting the HLDA signal. Once the HLDA signal goes high, the DMA controller activates the DACK signal to the requesting peripheral and gains the control of the system bus. The DMA controller is the sole master of the bus, till the DMA operation is over. The CPU remains in the HOLD status (all of its signals are tristated except HOLD and HLDA), till the DMA controller is the master of the bus. In other words, the DMA controller interfacing circuit implements a switching arrangement for the address, data and control busses of the memory and peripheral subsystem from/to the CPU to/from the DMA controller. A conceptual implementation of the system is shown in Fig. 7.7(a).



**Fig. 7.7(a)** Interfacing a Typical DMA Controller with a System

To explain the interfacing of 8257 with 8086 let us consider an interfacing example.

#### Problem 7.1

Interface DMA controller 8257 with 8086 so that the channel 0 DMA address register has an I/O address 80H and the mode set register has an address 88H. Initialize the 8257 with normal priority, TC stop and non-extended write. Autoload is not required. The transfer is to take place using channel 0. Write an ALP to move 2KB of data from a peripheral device to memory address 2000:5000H, with the above initialisation.

**Solution** Figure 7.7(b) shows interfacing connections of DMA controller 8257 with 8086.

As the DMA controller can generate only 16-bit address, while the CPU generates 20-bit address, the four upper address bits are generated and latched on the bus externally using a latch 8212. The 8086 uses three more 8212 latches and two 74245 buffers to demultiplex the address and data buses. These latches are controlled using the AEN signal of the DMA controller. If the DMA controller is in master mode, these will be automatically disabled and if the DMA controller is in slave mode, i.e.

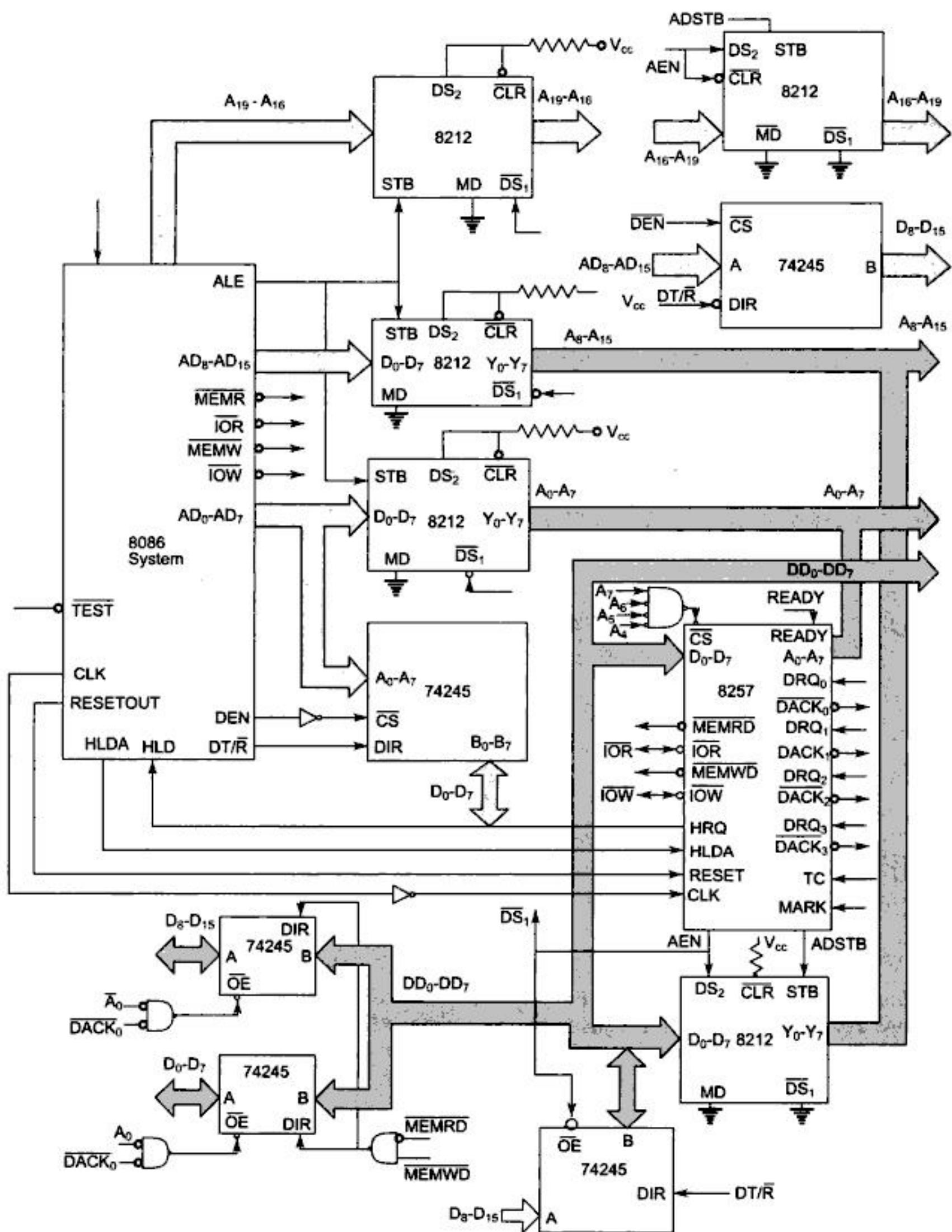


Fig. 7.7(b) Interfacing 8257 with 8086

the buses are in the control of the CPU, these latches are enabled, using the  $\overline{DS_1}$  signal. The data bus  $D_0-D_{15}$  is the general system data bus while the bus  $DD_0-DD_7$  is a data bus to be used by an 8-bit peripheral that works under the control of the DMA controller. The 8-bit data bus  $DD_0-DD_7$  is generated from the system data bus  $D_0-D_{15}$  using two additional data buffers which enable the 8-bit peripheral to access the even as well as odd addressed memory locations over the 8-bit data bus  $DD_0-DD_7$ . The MEMRD and MEMWD signals decide the direction of data flow under the control of the DMA controller. The  $A_0$  and  $DACK_0$  signals enable the two buffers only for the DMA controlled data transfer on channel 0. The DMA controller requires an additional latch to demultiplex the address bus  $A_8-A_{15}$  from the data bus  $D_0-D_7$ , generated by it. This latch is enabled by the ADSTB signal generated by the DMA controller. An additional 74245 is used to read and write the DMA controller registers under the control of the CPU. The inverted clock delays the DMA controller operation as compared to the CPU. Note that the upper data bus  $D_8-D_{15}$  is used for initialising the DMA controller in the slave mode. Also note the corresponding 16-bit instructions used to initialise the 8-bit peripheral using the upper 8-bit data bus  $D_8-D_{15}$ . Note the circuit arrangements made for accessing the even as well as odd addresses using  $D_0-D_7$ . All the initialisation command words should be derived before writing the program.

**Mode Set Register** As per the problem specification, we need the following: enable TC stop, enable channel 0, disable auto-load, Disable extended-write, disable rotating priority, disable all other channels. As already discussed, the individual bits of the mode set register are set or reset as shown below.

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	
0	1	0	0	0	0	0	1	= 41 H

**DMA Address Register** The DMA address register contains the starting address of the memory block which is to be accessed using DMA, i.e. 5000H.

**Terminal Count Register** As has been mentioned in Section 7.1.1, the last significant 14-bits of this register will contain the binary equivalent of the required number of the DMA cycles, i.e. number of bytes to be transferred minus one. As per the requirement, 2Kbytes of data have to be transferred from the device to memory. Therefore, the low order 14-bits of TC register will contain 7FFH. Moreover, the DMA operation in this case is going to be a memory write operation, hence  $A_{15}$  and  $A_{14}$  of this register should be 0 and 1. The TC register contents for these specifications are shown below.

A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	= 47FFH	

The ALP for Problem 7.1 is given as follows.

```

ASSUME      CS:CODE, DS:DATA
CODE        SEGMENT
START:      MOV AX, DATA          ; Initialize Data Segment
            MOV DS, AX          ;
            MOV AX, DMAL         ; Load DMA address register with
            OUT 80H, AX         ; lower byte of DMA address
            MOV AX, DMAH         ; Load higher byte of DMA address
            OUT 80H, AX         ; register of Channel 0
            MOV AX, TCL          ; Load lower byte TC register of
            OUT 81H, AX          ; channel 0
            MOV AX, TCH          ; Load higher byte of TC register
            OUT 81H, AX          ;

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Mode Register** Each of the DMA channel has an 8-bit mode register. This is written by the CPU in program mode. Bits 0 and 1 of the mode register determine which of the four channel mode registers is to be written. The bits 2 and 3 indicate the type of DMA transfer. As we have discussed earlier, there are three types of DMA transfer, viz. memory read, memory write and verify transfer. Bit 4 of the mode register indicates whether auto-initialization is selected or not, while bit 5 indicates whether address increment or address decrement mode is selected. The definition of the mode register is presented in Fig. 7.10.

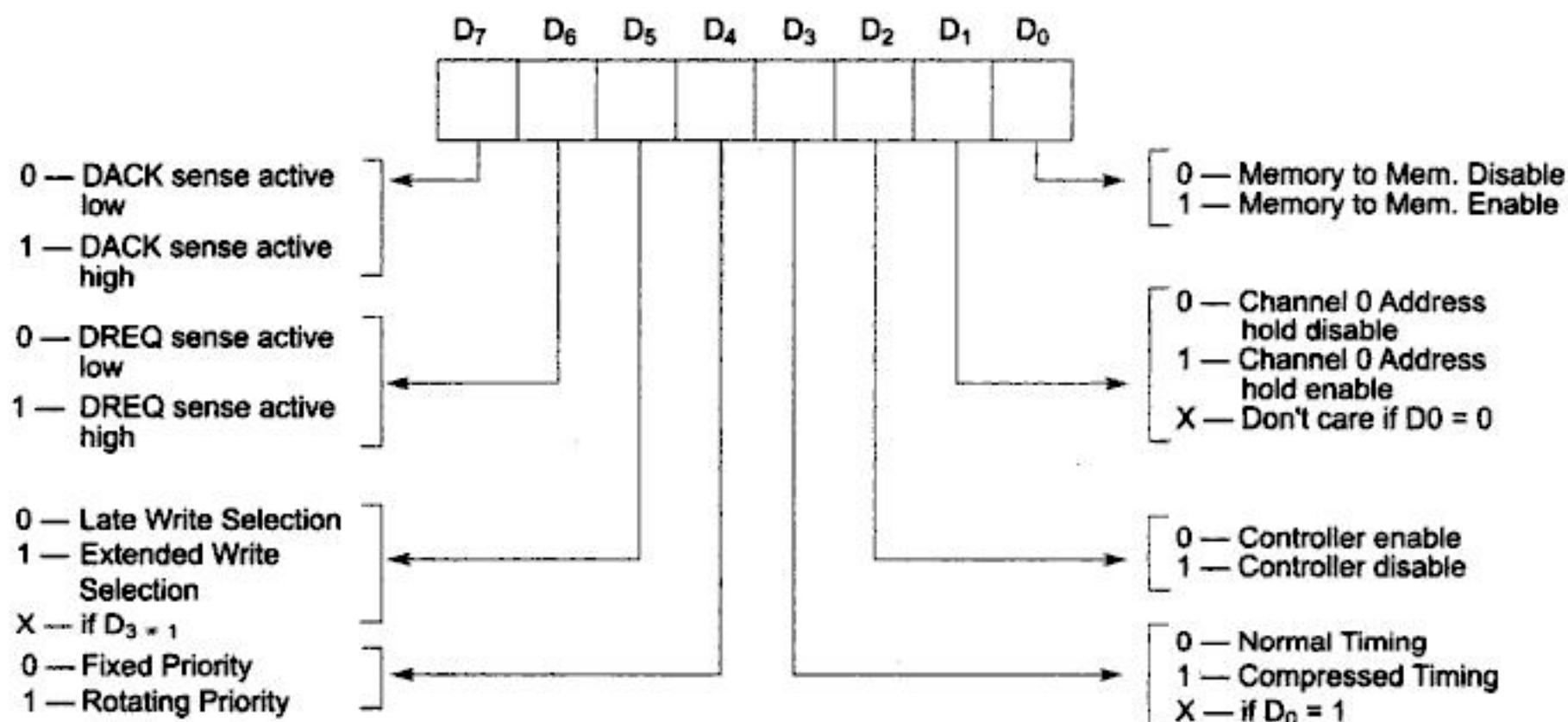


Fig. 7.9 Command Register Definition

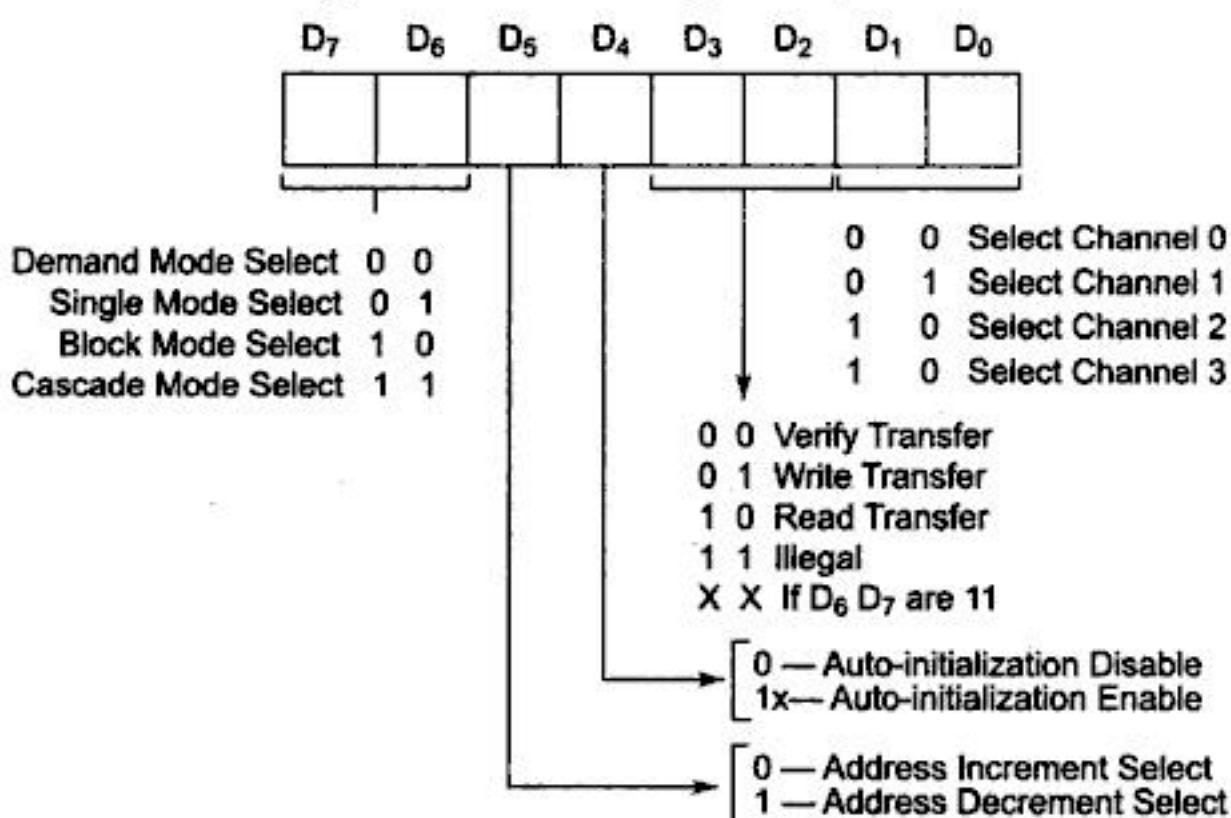


Fig. 7.10 Mode Register Definition



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Status Register** The status register keeps the track of all the DMA channel pending requests and the status of their terminal counts. The bits D<sub>0</sub>-D<sub>3</sub> are updated (set) every time, the corresponding channel reaches TC or an external EOP occurs. These are cleared upon reset and also on each status read operation. Bits D<sub>4</sub>-D<sub>7</sub> are set, if the corresponding channels request services. Figure 7.13 shows the definition of the status register.

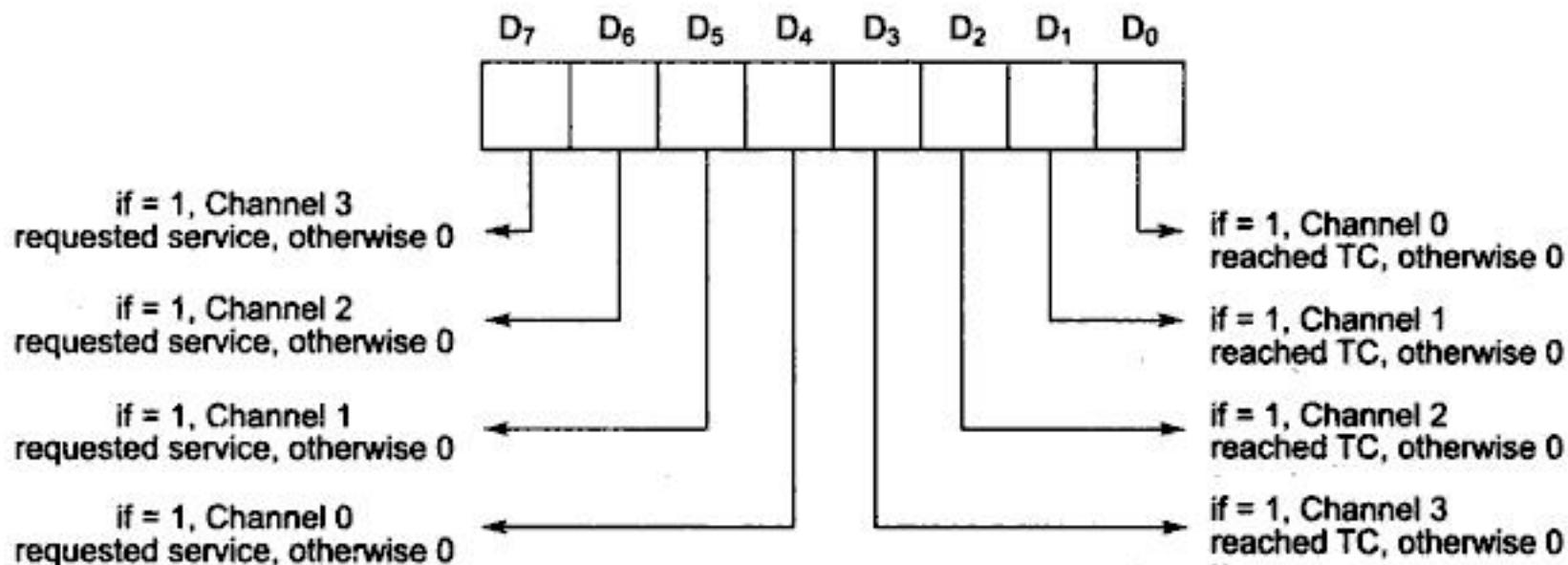


Fig. 7.13 Status Register Definition

### 7.3.3 Signal Descriptions of 8237

Figure 7.14 shows the pin diagram of 8237. The functional signal description of each pin is discussed in brief as follows:

**V<sub>cc</sub>** This is a +5V supply pin required for operation of the circuit.

**GND** This is a return line for the supply (ground pin of the IC).

**CLK** This is the internally required CLK signal for deriving the internal timings required for the circuit operation.

**CS** This is an active-low chip select input of the IC.

**RESET** A high on this input line clears the command, status, request and temporary registers. It also clears the internal first/last flip-flop and sets the mask register. The 8237 remains stuck to reset till the RESET pin is high.

**READY** This active-high input is used to match the read or write speed of 8237 with slow memories or I/O devices.

IOR	1	40	A <sub>7</sub>
IOW	2	39	A <sub>6</sub>
MEMR	3	38	A <sub>5</sub>
MEMW	4	37	A <sub>4</sub>
*	5	36	EOP
READY	6	35	A <sub>3</sub>
HLDA	7	34	A <sub>2</sub>
ADSTB	8	33	A <sub>1</sub>
AEN	9	32	A <sub>0</sub>
HRQ	10	8237	V <sub>cc</sub>
CS	11	30	DB <sub>0</sub>
CLK	12	29	DB <sub>1</sub>
RESET	13	28	DB <sub>2</sub>
DACK2	14	27	DB <sub>3</sub>
DACK3	15	26	DB <sub>4</sub>
DRQ <sub>3</sub>	16	25	DACK0
DRQ <sub>2</sub>	17	24	DACK1
DRQ <sub>1</sub>	18	23	DB <sub>5</sub>
DRQ <sub>0</sub>	19	22	DB <sub>6</sub>
GND	20	21	DB <sub>7</sub>

\* Pin 5 should be always at logic high. An internal pull-up pulls it up when floating else it should be tied to V<sub>cc</sub>.

Fig. 7.14 Pin Diagram of 8237



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Table 7.4(a) (Contd.)**

N	Number	N stands for the number of data bytes written in a Sector.
NCN	New Cylinder Number	NCN stands for new Cylinder number which is going to be reached as a result of the Seek operation, i.e. desired position of Head.
ND	Non DMA Mode	ND stands for the operation in the Non-DMA Mode.
PCN	Present Cylinder Number	PCN stands for the Cylinder number at the completion of SENSE INTERRUPT STATUS Command Position of head at a present time.
R	Record	R stands for the Sector number, which will be read or written.
R/W	Read/Write	R/W stands for either Read (R) or Write (W) signal.
SC	Sector	SC indicates the number of Sectors per Cylinder.
SK	Skip	SK stands for Skip Deleted Data Address Mark.
SRT	Step Rate Time	SRT stands for the Stepping Rate for the FDD (1 to 16 ms in 1 ms increment). The same Stepping Rate applies to all drives (F = 1 ms, E = 2 ms, etc.).
ST <sub>0</sub>	Status 0	ST <sub>0-3</sub> stand for one of the four registers which store the status information after a command has been executed. This information is available during the result phase after command execution.
ST <sub>1</sub>	Status 1	
ST <sub>2</sub>	Status 2	
ST <sub>3</sub>	Status 3	These register should not be confused with the main status registers (selected by A <sub>0</sub> = 0). ST <sub>0-3</sub> may be read only after a command has been executed and contain information relevant to that particular command.
STP	Step	During a Scan operation, STP = 1, the data in contiguous sectors is compared byte by byte with data sent from the processor (or DMA), and if STP = 2, then alternate sectors are read and compared.

**Table 7.4(b) Command Sheet of 8272**

Phase	R/W	Data Bus								Remarks
		D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>		
READ DATA										
Command	W	MT	MFM	SK	0	0	1	1	0	Command Codes
	W	0	0	0	0	0	HDS	DS <sub>1</sub>	DS <sub>0</sub>	
	W	_____	_____	_____	C	_____	_____	_____	_____	Sector ID information
	W	_____	_____	_____	H	_____	_____	_____	_____	prior to Command
	W	_____	_____	_____	R	_____	_____	_____	_____	execution
	W	_____	_____	_____	N	_____	_____	_____	_____	
	W	_____	_____	_____	EOT	_____	_____	_____	_____	
	W	_____	_____	_____	GPL	_____	_____	_____	_____	
	W	_____	_____	_____	DTL	_____	_____	_____	_____	

(Contd.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Table 7.4(b) (Contd.)**

	W	_____	R	_____	execution
	W	_____	N	_____	
	W	_____	EOT	_____	
	W	_____	GPL	_____	
	W	_____	DTL	_____	
Execution					Data transfer between the FDD and main system
Result	R	_____	ST <sub>0</sub>	_____	Status information after Command execution
	R	_____	ST <sub>1</sub>	_____	
	R	_____	ST <sub>2</sub>	_____	
	R	_____	C	_____	
	R	_____	H	_____	Sector ID Information after command execution
	R	_____	R	_____	
	R	_____	N	_____	

WRITE DELETED DATA										
Command	W	MT	MFM	0	0	1	0	0	1	Command Codes
	W	0	0	0	0	0	HDS	DS <sub>1</sub>	DS <sub>0</sub>	
	W	_____				C	_____			Sector ID information prior to Command execution
	W	_____				H	_____			
	W	_____				R	_____			
	W	_____				N	_____			
	W	_____				EOT	_____			
	W	_____				GPL	_____			
	W	_____				DTL	_____			
Execution										Data transfer between the FDD and main system
Result	R	_____				ST <sub>0</sub>	_____			Status information after Command execution
	R	_____				ST <sub>1</sub>	_____			
	R	_____				ST <sub>2</sub>	_____			
	R	_____				C	_____			
	R	_____				H	_____			Sector ID Information after command execution
	R	_____				R	_____			
	R	_____				N	_____			

(Contd.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Table 7.10** (Contd.)

D <sub>3</sub>	Not Ready	NR	When the FDD is the not-ready state and a read or write command is issued, this flag is set. If a read or write command is issued to Side 1 of a single sided drive, then this flag is set.
D <sub>2</sub>	Head Address	HD	This flag is used to indicate the state of the head at Interrupt.
D <sub>1</sub>	Unit Select 1	US <sub>1</sub>	These flags are used to indicate a
D <sub>0</sub>	Unit Select 0	US <sub>0</sub>	Drive Unit Number at Interrupt

**STATUS REGISTER 1**

D <sub>7</sub>	End of Cylinder	EN	When the FDC tries to access a sector beyond the final sector of a cylinder, this flag is set.
D <sub>6</sub>			Not used. This bit is always 0.
D <sub>5</sub>	Data Error	DE	When the FDC detects a CRC error in either the ID field or the data field, this flag is set.
D <sub>4</sub>	Over Run	OR	If the FDC is not serviced by the main system during data transfers, within a certain the interval, this flag is set.
D <sub>3</sub>	.		Not used. This bit is always 0.
D <sub>2</sub>	No Data	ND	During execution of READ DATA, WRITE DELETED DATA or SCAN command, if the FDC cannot find the sector specified in the IDR register, this flag is set. During executing the READ ID command, if the FDC cannot read the ID field without an error, this flag is set. During executing the READ A CYLINDER command, if the starting sector cannot be found, this flag is set.

D <sub>1</sub>	Not Writable	NW	During execution of WRITE DATA, WRITE DELETED DATA or FORMAT A CYLINDER command. If the FDC detects a write protect signal from the FDD, then this flag is set.
----------------	--------------	----	---

D <sub>0</sub>	Missing Address Mark	MA	If the FDC cannot detect the ID address mark after encountering the index hole twice, this flag is set. If the FDC cannot detect the Data Address mark or Deleted Data Address mark, this flag is set. Also at the same time, the MD (Missing Address Mark in Data Field) of Status Register 2 is set.
----------------	----------------------	----	---

**STATUS REGISTER 2**

D <sub>7</sub>		Not used. This bit is always 0.
----------------	--	---------------------------------

(Contd.)

**Table 7.10 (Contd.)**

D <sub>6</sub>	Control Mark	CM	During executing the READ DATA or SCAN Command, if the FDC encounters a sector which contains a Deleted Data Address mark, this flag is set.
D <sub>5</sub>	Data Error Data Field	DD	If the FDC detects a CRC error in the data field then this flag is set.
D <sub>4</sub>	Wrong Cylinder	WC	This bit is related with the ND bit, and when the contents of C on the medium is different from that stored in the IDR, this flag is set.
D <sub>3</sub>	Scan Equal Hit	SH	During execution of the SCAN command. If the condition of "equal" is satisfied, this flag is set.
D <sub>2</sub>	Scan Not Satisfied	SN	During executing the SCAN command, if the FDC cannot find a sector on the cylinder which meets the conditions, then this flag is set.
D <sub>1</sub>	Bad Cylinder	BC	This bit is related with the ND bit, and when the content of C on the medium is different from that stored in the IDR and the content of C is FF, then this flag is set.
D <sub>0</sub>	Missing Address Mark in Data Field	MD	When data is read from the medium, if the FDC cannot find a Data Address Mark or Deleted Data Address Mark, then this flag is set.

**STATUS REGISTER 3**

D <sub>7</sub>	Fault	FT	This bit is used to indicate the status of the Fault Signal from the FDD.
D <sub>6</sub>	Write Protected	WP	This bit is used to indicate the status of the Write Protected signal from the FDD.
D <sub>5</sub>	Ready	RDY	This bit is used to indicate the status of the Ready signal from the FDD.
D <sub>4</sub>	Track 0	T0	This bit is used to indicate the status of the Track 0 signal from the FDD.
D <sub>3</sub>	Two Side	TS	This bit is used to indicate the status of the Two Side signal from the FDD.
D <sub>2</sub>	Head Address	HD	This bit is used to indicate the status of the Side Select signal to the FDD.
D <sub>1</sub>	Unit Select 1	US1	This bit is used to indicate the status of the Unit Select 1 signal to the FDD.
D <sub>0</sub>	Unit Select 0	US <sub>0</sub>	This bit is used to indicate the status of the Unit Select 0 signal to the FDD.

**Problem 7.4**

Write an ALP to initialize the 8237 and 8272 combination to transfer data to a FDD under the following conditions; the 8237 is set to transfer from memory to a peripheral using channel 0 masking all other channels. It is initialized for normal timing, fixed priority, extended write with DREQ active-high and DACK active low. The 8237 should work in auto-initialization mode with address increment and single byte transfer mode. The addresses of the internal registers of 8237 are given as follows:

Command Register-F8H

Mode Set Register-FBH

Request Register-F9H

Mask Register-FA/FFH

Status Register-F8H

Temporary Register-FDH

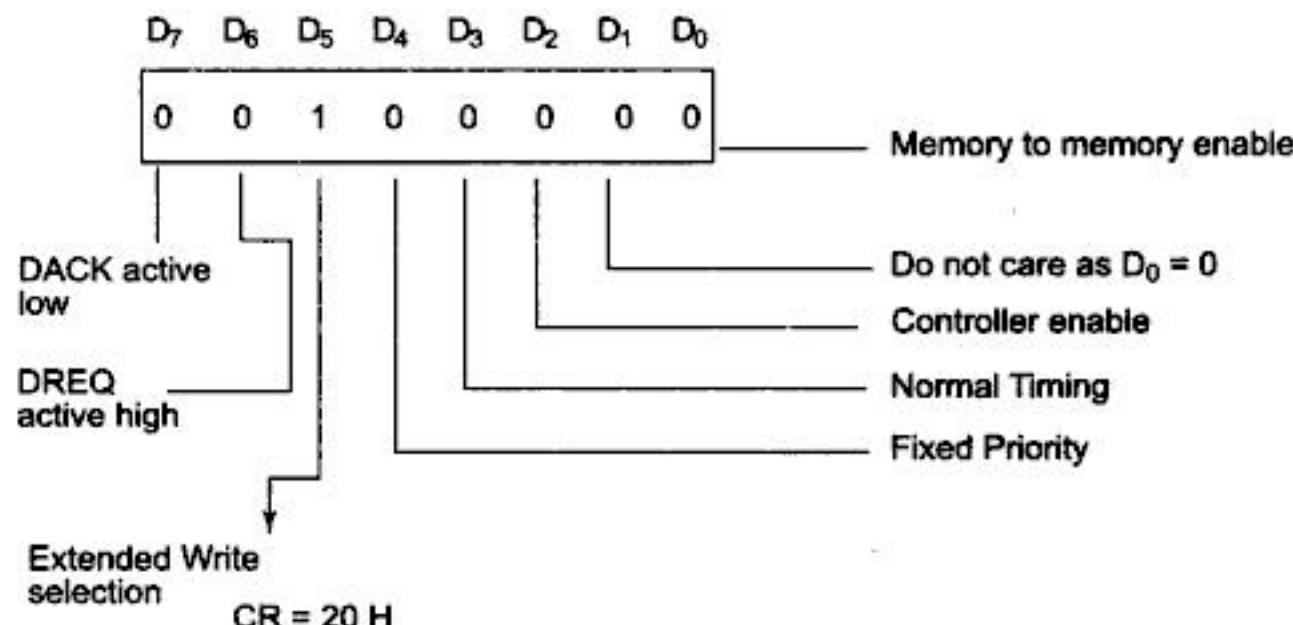
Byte Pointer F/F-FCH

Base and Current Address Register of Channel 0-E0H

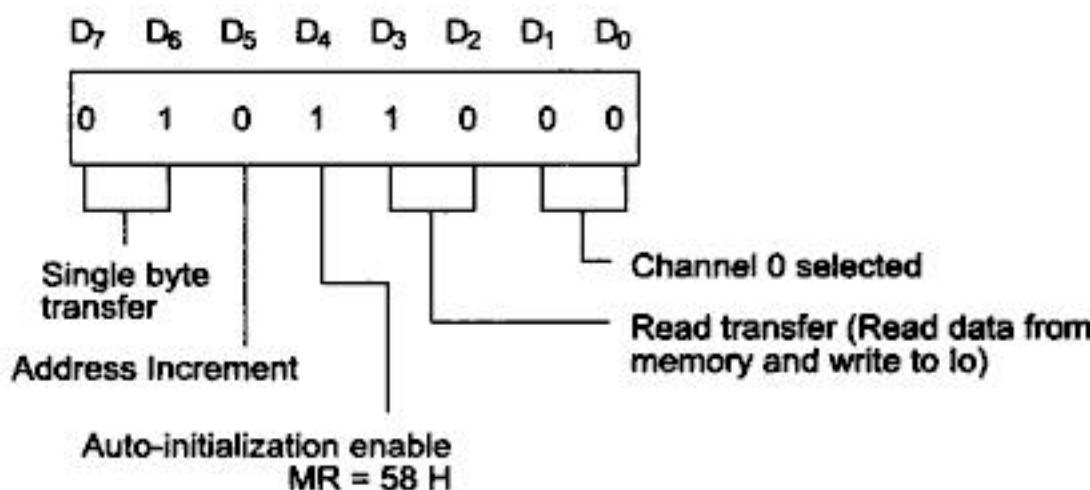
Base and Current Word Count Register of Channel 0-E1H

Use Drive 0 of 8272 to accomplish the transfer to FDD. Set 8272 in DMA mode with Head Unload Time 16 ms, Step Rate Time 1 ms, Head Load Time 2 ms. The data block is of 4 Kb size. It is to be transferred in MFM mode using 3 bytes/sector, MT format. Assume that the disk is already formatted.

**Solution** The command words of 8237 are determined as given.



The mode register contents are decided as given:





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Line Number		Line Counter Mode 0	Line Counter Mode 1
0	□ □ □ □ □ □ □ □ □	0 0 0 0	1 0 1 1
1	□ □ □ □ □ □ □ □ □	0 0 0 1	0 0 0 0
2	□ □ □ □ □ □ □ □ □	0 0 1 0	0 0 0 1
3	□ □ □ □ □ □ □ □ □	0 0 1 1	0 0 1 0
4	□ □ □ □ □ □ □ □ □	0 1 0 0	0 0 1 1
5	□ □ □ □ □ □ □ □ □	0 1 0 1	0 1 0 0
6	□ □ □ □ □ □ □ □ □	0 1 1 0	0 1 0 1
7	□ □ □ □ □ □ □ □ □	0 1 1 1	0 1 1 0
8	□ □ □ □ □ □ □ □ □	1 0 0 0	0 1 1 1
9	□ □ □ □ □ □ □ □ □	1 0 0 1	1 0 0 0
10	□ □ □ □ □ □ □ □ □	1 0 1 0	1 0 0 1
11	□ □ □ □ □ □ □ □ □	1 0 1 1	1 0 1 0

Top and Bottom Lines are Blanked

Fig. 7.25(a) Underline in Line 10

Line Number		Line Counter Mode 0	Line Counter Mode 1
0	□ □ □ □ □ □ □ □ □	0 0 0 0	0 1 1 1
1	□ □ □ □ □ □ □ □ □	0 0 0 1	0 0 0 0
2	□ □ □ □ □ □ □ □ □	0 0 1 0	0 0 0 1
3	□ □ □ □ □ □ □ □ □	0 0 1 1	0 0 1 0
4	□ □ □ □ □ □ □ □ □	0 1 0 0	0 0 1 1
5	□ □ □ □ □ □ □ □ □	0 1 0 1	0 1 0 0
6	□ □ □ □ □ □ □ □ □	0 1 1 0	0 1 0 1
7	□ □ □ □ □ □ □ □ □	0 1 1 1	0 1 1 0

Top and Bottom Lines are Blanked

Fig. 7.25(b) Underline in Line 7

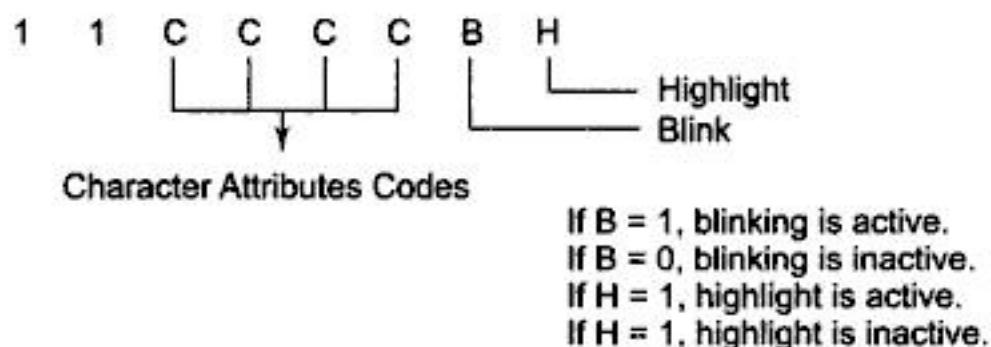
**Raster Timings** The CCLK input drives a character counter that counts the characters being displayed (from 1 to 80). The character counter output drives a line counter that starts counting line retrace intervals (from 2 to 32 for 1 to 16 lines respectively). The line counter generates line address outputs ( $LC_0 - LC_3$ ) for the character generator. After all the lines programmed are counted by the line counter, it increments the row counter and resets the character counter and line counter again. The row counter is driven by the line counter and controls the row buffers and counts the number of rows displayed. After all the rows in a frame (from 1 to 64) are counted, it starts counting the vertical retrace interval (programmable from 1 to 4). The VSP output is active during the horizontal and vertical retrace intervals. The dot level timing circuit must synchronize these outputs with the video signal for driving the CRT display.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

( $LA_0-LA_1$ ), Video Suppression Output (VSP) and the Light Enable Output (LTEN). The dot level timing circuitry can use these signals to generate the proper symbols. The character attributes can be programmed to blink or highlight the characters individually. Blinking is accomplished using the Video Suppression Output (VSP). The blink frequency is equal to the screen refresh frequency divided by 32. The highlighting is accomplished by activating the HGLT output.

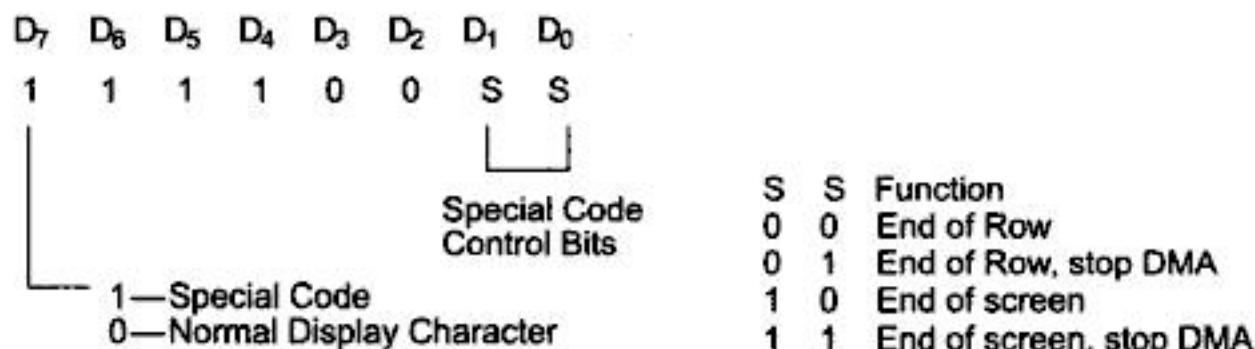
### Character Attributes



**Fig. 7.27**

As shown in Fig. 7.27, character attribute codes 1101, 1110 and 1111 are illegal, while 1011 code is not recommended. 1100 is for special codes. Each of the other combinations represent a graphical symbol. The detailed description of these symbols cannot be presented here. Intel's literature may be referred to for detailed description of these attributes.

**Special Codes** Four special codes are designed to reduce the required memory, software and DMA overheads. The special control code characters are shown in the special code byte format as shown in Fig. 7.28.



**Fig. 7.28**

The end of row code activates VSP and holds it at the end of the line. The end of row stop DMA code causes the on-going DMA cycle to be terminated for the rest of the row, if it is written to the row buffer. The end of screen code activates VSP and holds it at the end of the frame. The end of screen stop DMA code causes the DMA to be terminated for the rest of the frame, if written to the row buffer. If the stop DMA code is not the last code in a burst or a row, the DMA cycle is not stopped until the next character is read. In this case, a dummy character must be placed in memory after the stop DMA character.

**Field Attributes** The field attribute code is an 8-bit code, where each bit is a control bit which affects the visual characteristics for a single character or a field (string) of characters. In case of a string of characters, the visual characteristics of these characters may be changed like highlighted, blinked, etc. using the field attribute codes. The visual characteristics of all the characters will be changed



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Table 7.12 Reset Command and Parameters**

<i>Operation</i>	<i>A<sub>0</sub></i>	<i>Description</i>	<i>D<sub>7</sub></i>	<i>D<sub>6</sub></i>	<i>D<sub>5</sub></i>	<i>D<sub>4</sub></i>	<i>D<sub>3</sub></i>	<i>D<sub>2</sub></i>	<i>D<sub>1</sub></i>	<i>D<sub>0</sub></i>
Command	Write	1 Rest	0	0	0	0	0	0	0	0
Parameters	Write	0 Screen Composition Byte 1	S	H	H	H	H	H	H	H
	Write	0 Screen Composition Byte 2	V	V	R	R	R	R	R	R
	Write	0 Screen Composition Byte 3	U	U	U	L	L	L	L	L
	Write	0 Screen Composition Byte 4	M	F	C	C	Z	Z	Z	Z

The symbols in Table 7.12 are explained in details, in the following Tables 7.13 to 7.22.

**Table 7.13 S Bit Definition**

<i>Spaced Rows S</i>	<i>Function</i>
0	Normal Rows
1	Spaced Rows

**Table 7.14 HHHHHHHH Bit Definitions**

<i>Parameter HHHHHHHH</i>	<i>No. of Horizontal Characters/Row</i>
00000000	1
00000001	2
00000010	3
⋮	⋮
1001111	80
1010000	Undefined
⋮	⋮
1111111	Undefined



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



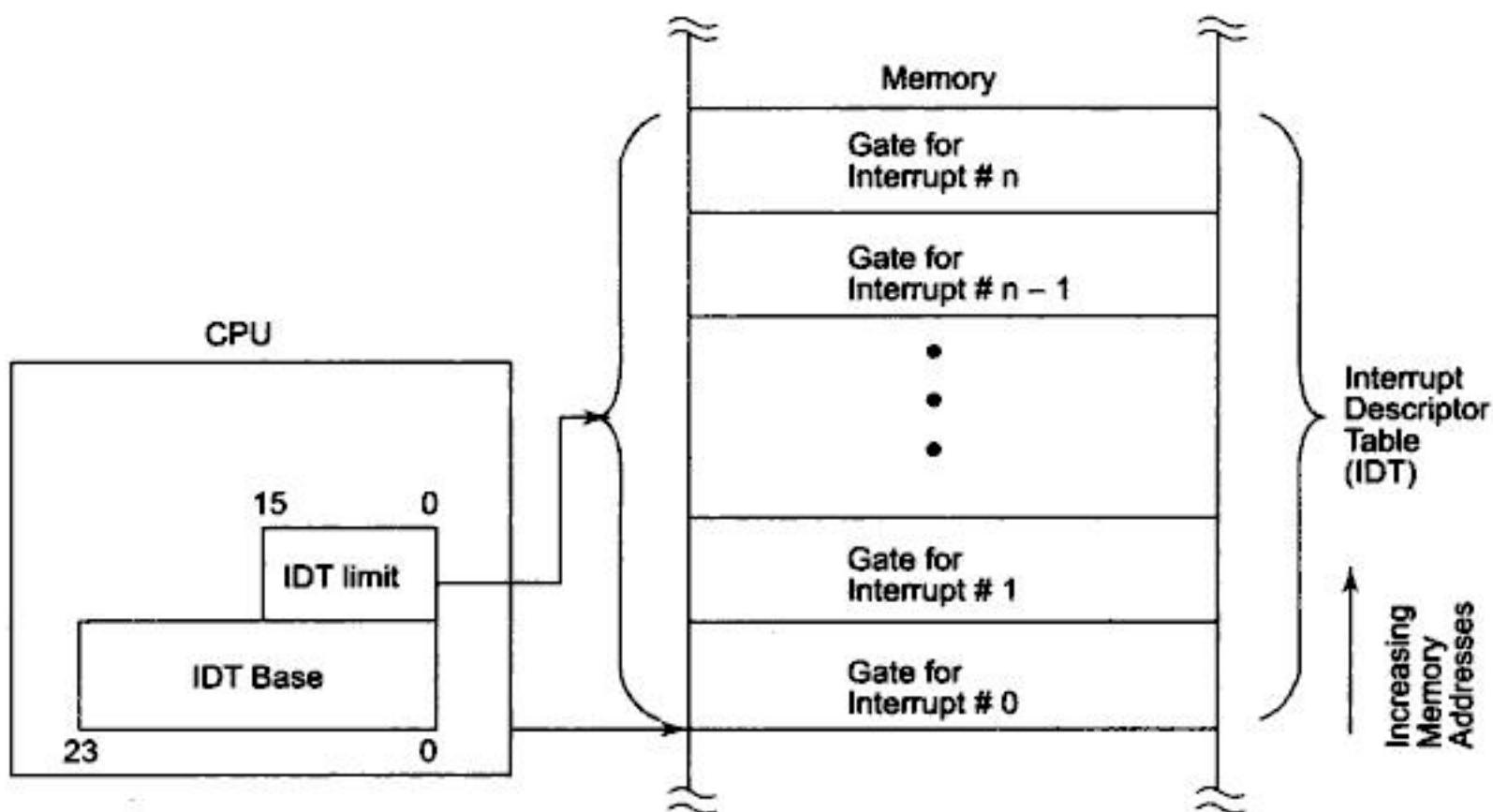
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



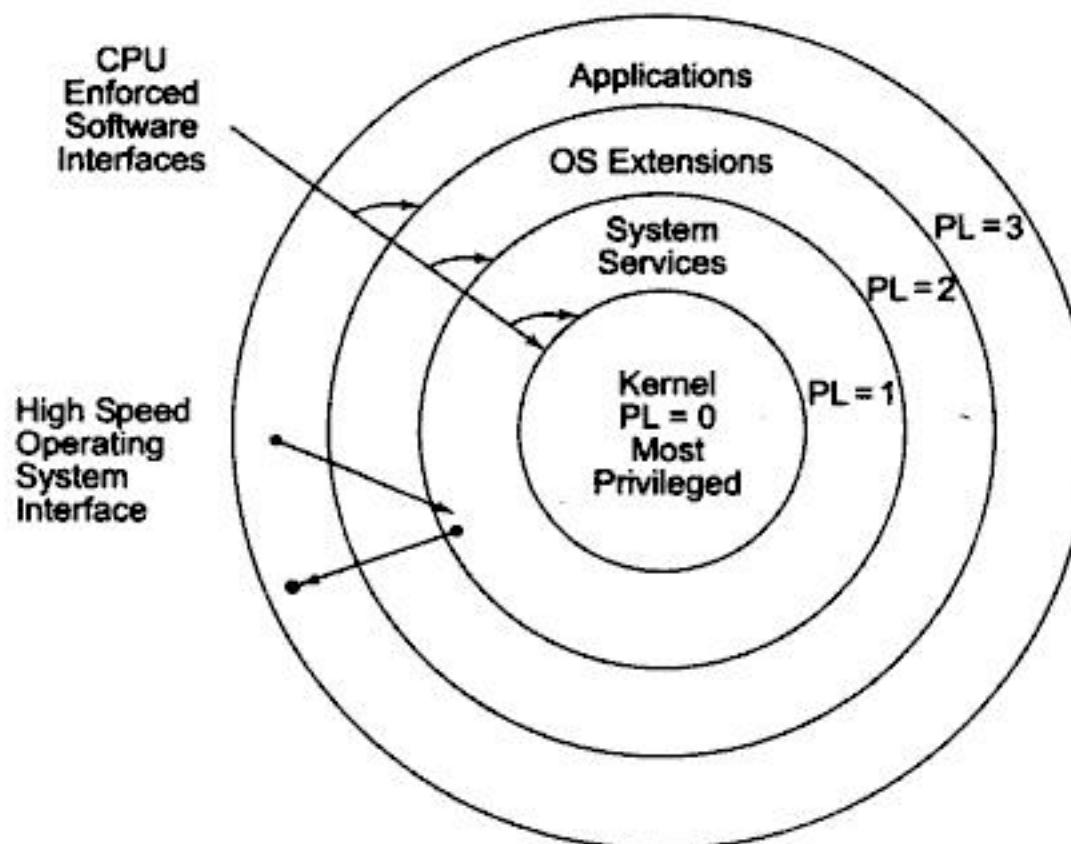
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Fig. 9.14** Interrupt Descriptor Table Organisation



Note: PL Becomes Numerically Lower as Privilege Level Increases

**Fig. 9.15** Four Level Privilege Mechanism

### 9.6.1 Task Privilege

Each task is assigned a privilege level, which indicates the priority or privilege of that task. Any one of the four privilege levels may be used to execute a task. The task privilege level at that instant is called



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



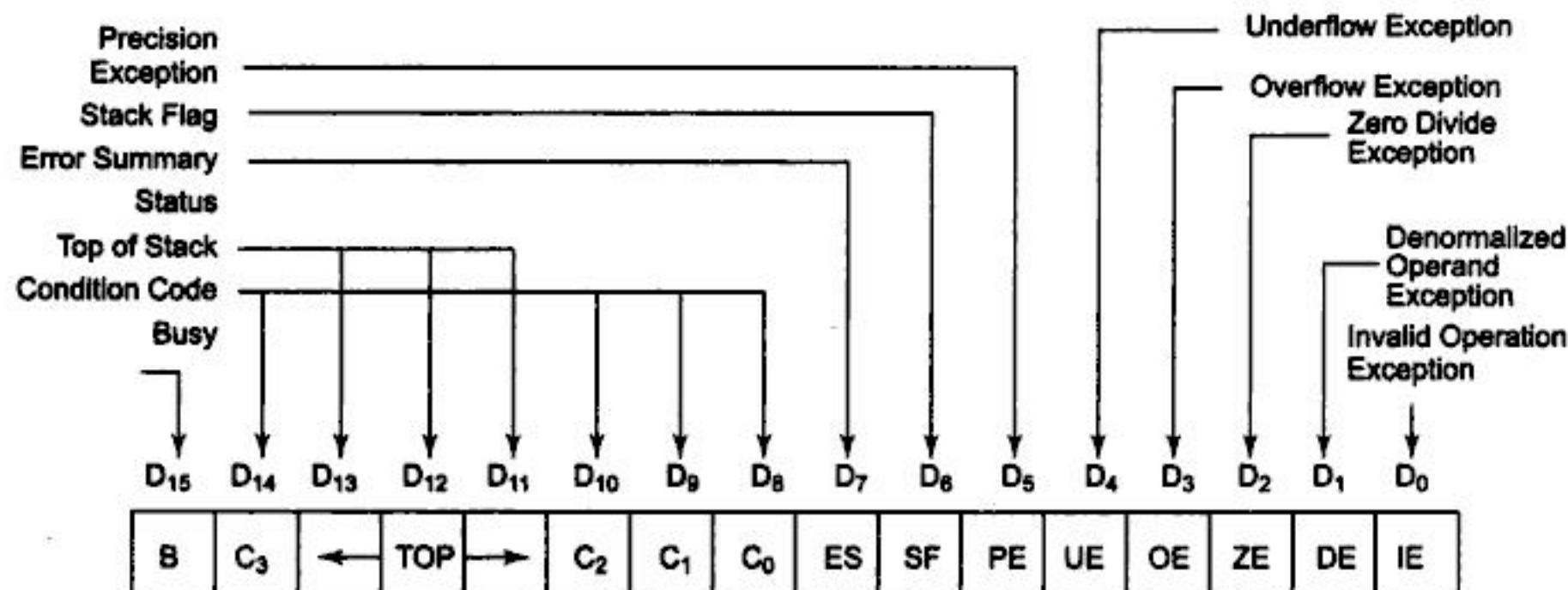
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Exception Flags ( $D_5-D_0$ )** These exception flags are described in the Fig. 9.31. These are used to show the generation of an exception while 80287 is executing.

The status word of 80287 is shown in the Fig. 9.32. This can also be written by using FLDENV or FRSTOR instructions.



**Notes:**

1.  $ES$  is set if any unmasked exception bit is set, cleared otherwise.
2. See Table 8.1 for condition code interpretation.
3.  $TOP$  Values

000 = Register 0 is Top of Stack

001 = Register 1 is Top of Stack

•  
•  
•

111 = Register 7 is Top of Stack

**Fig. 9.32 Status Word of 80287 (Intel Corp.)**

**(b) Control Word** The control word is used to select one of the processing options amongst the ones provided by 80287. The various bits of the control word are discussed as follows:

**Masking Bits ( $D_0-D_5$ )** These are the six masking bits used to mask the six exceptions shown in the status register. If this is '1', the respective exception is masked.

**Precision Control bits ( $D_8-D_9$ )** These are used to set the internal precision of 80287.

These bits affect ADD, SUB, DIV, MUL and SQRT results. For other instructions the precision is decided by the opcode or the extended precision format.

**Rounding Control Bits ( $D_{10}-D_{11}$ )** These bits are used to set rounding or chopping, as specified in IEEE standard. Rounding control bits affect only the instructions which perform rounding after the operation, for example, in arithmetic and transcendental instructions.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

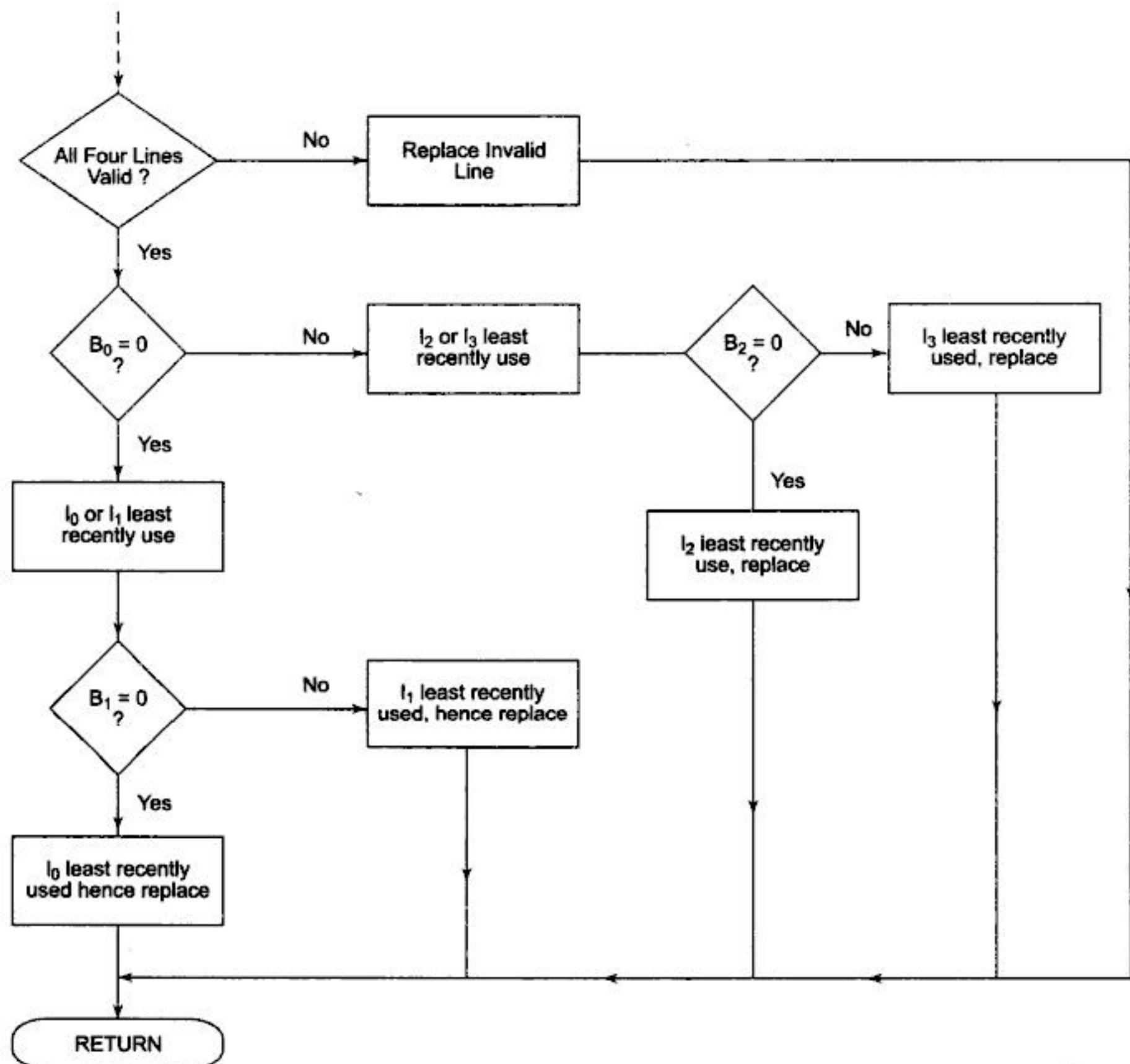


Fig. 10.20 LRU Replacement Algorithm

## Exercises

- 10.1 Enlist the salient features of 80386.
- 10.2 Draw and discuss internal architecture of 80386 in detail.
- 10.3 Explain the following signal functions of 80386.
 

(i) BE <sub>0</sub> #-BE <sub>3</sub> #	(ii) W/R#	(iii) D/C
(iv) ADS#	(v) NA#	(vi) BS <sub>16</sub>

- 10.4 Draw and discuss the register set of 80386 and explain a typical function of each of the registers in brief.

10.5 Draw and discuss the flag register of 80386 in detail.

10.6 Explain the use of each of the following registers of 80386.

(i) Segment Descriptor Registers      (ii) Control Registers

(iii) Debug and Test Registers      (iv) System Address Registers

10.7 Explain the different additional addressing modes supported by 80386 over 80286.

10.8 Enlist the different data types supported by 80386.

10.9 Explain the physical address formation in real address mode of 80386.

10.10 Explain the physical address formation in PVAM of 80386.

10.11 Draw and discuss the structures of the different descriptors supported by 80386.

10.12 What do you mean by a descriptor table?

10.13 How many maximum descriptors can be accessed by 80386 for a single task? What is the memory addressing capability of a single descriptor? Justify the virtual memory addressing capability of 80386 per task.

10.14 Discuss the different descriptor types supported by 80386.

10.15 What are the different exceptions generated by 80386?

10.16 What do you mean by paging? What are its advantages and disadvantages?

10.17 Draw and discuss the paging mechanism of 80386 in details.

10.18 What are the differences between the logical addresses, linear addresses and physical addresses?

10.19 Explain the procedure of converting a linear address into a physical address.

10.20 What is translation look aside buffer? How does it speed up the execution of the programs?

10.21 Write a short note on virtual 8086 mode of 80386.

10.22 Draw and discuss the architecture of 80387.

10.23 Enlist the different data types supported by 80387.

10.24 Discuss the register set of 80387.

10.25 Draw and discuss the interface of 80387 with 80386.

10.26 Enlist the salient features of 80486.

10.27 Draw and discuss the flag register of 80486.

10.28 Enlist four major architectural advancement in 80486 over 80386.

10.29 What is the use of TEST and DEBUG facility in 80486?

10.30 What do you mean by cache memory? How does it speed up the program execution?

10.31 Explain the cache management unit of 80486.

10.32 Write short notes on the following.

(i) Cache Maintenance Operations      (ii) Paging Unit

10.33 Enlist the data types supported by 80486.

10.34 Enlist the different functional groups of signals provided by 80486.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

from the complexity in the addressing modes and also in the instruction set of the conventional CISC CPUs like 386 or 486, there was not much parallelism (pipelining) in these processors, which resulted in a comparatively slower speed compared to the RISC based CPUs. One of the major bottlenecks in case of the earlier CPUs was possibly the inclusion of a separate math coprocessor. The 8087, math coprocessor which is a companion of 8086 executes a floating-point instruction, whenever it comes across such an instruction. The architecture of these math coprocessors have continuously evolved from 8087 to 80387, the companion processor of 80386 CPU.

Interestingly, the instruction and data transfer between 80386 and 80387 takes place through an I/O handshaking mode. The 80386 requires around 15 clock cycles to perform the I/O handshaking with 80387 and to take care of some internal housekeeping operations. Thus, as we can see that even if the operating speed of these math co-processors are increased, the overall floating-point speed performance does not increase appreciably.

This was the reason, why an on-chip coprocessor was included while designing the 80486 CPU. One may note that 80486 requires only about four clock cycles for floating-point transactions, resulting in a better floating-point performance. Another limitation of this coprocessor architecture is that the total number of internal registers is less, i.e. only eight in case of 8087/387 or even 80487 CPU. These are essentially stack oriented, rather than register oriented processors, which restrict their speed. There are however other floating-point processors from other vendors (say Weitek math coprocessor) which can perform single-precision floating-point operations in a single cycle or double-precision in two cycles, etc. These math coprocessors have more number of internal registers, and are essentially based on register oriented architecture. All that we wanted to convey is the fact that floating-point operation was really a major bottleneck in the CPUs like 286, 386, or 486.

Keeping in view the above scenario, we will now present the architecture of Pentium which was an extremely challenging attempt to bridge the gap between CISC based low-end PCs and RISC based high-end workstations.

### 11.1 SALIENT FEATURES OF 80586 (PENTIUM)

In the introductory note we have hinted that the designers of Pentium had basically two clear points in mind:

- To design a CPU with enhanced complex instruction sets, which should remain code compatible with earlier X86 CPUs—from 8086 to 486 and,
- To achieve performance so as to match the third generation RISC performances.

Both these objectives were, to a large extent met while designing the Pentium CPU. Thus Pentium designers introduced a lot of RISC features while retaining the complex instruction sets supported by the earlier X86 CPUs.

A salient feature of Pentium is its *superscalar, superpipelined* architecture. It has two integer pipelines U and V, where each one is a 4-stage pipeline. This enhances the speed of integer arithmetic of Pentium to a large extent. Moreover, it has an on-chip floating-point unit, which has increased the floating-point performance manifold compared to the floating-point performances of 80386/486 processors.

Another feature of Pentium is that it contains two separate caches, viz. data cache and instruction cache. One may recall that in 80486 there was a single unified data/instruction cache. All these features will be explained in detail later in this section.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- (vii) Microcoding not required: Unlike in a CISC machine, in RISC architecture, instruction microcoding is not required. This is because of the availability of a set of simple instructions, and simple instructions may be easily built into the hardware.
- (viii) Load and Store architecture: The RISC architecture is primarily a Load and Store architecture, implying that all the memory accesses take place using Load or Store type operations.

In the next section we will investigate some of the critical issues involved in the design of RISC Architecture.

### 13.5 DESIGN ISSUES OF RISC PROCESSORS

In this section we will briefly discuss some of the important issues involved in the design of RISC based processors.

#### 13.5.1 Register Windowing

The concept of register windowing involves a mechanism where the chips expose 32 registers to the programmer at any one time, but these registers are just a “window” into a larger set of physical registers. The additional registers are hidden from view until you call a subroutine. For example other processor would push parameters on a stack for the called routine to pop off, SPARC processors just “rotate” the register window to give the called routine a fresh set of registers. The old window and the new window overlap, so that some registers are shared. As long as you’re careful about placing parameters in the right registers, the windows are a slick way to pass operands without using the stack at all.

Slick as it seems, registers windows have their drawback. The concept has been around for decades, yet SPARC is almost the only CPU architecture to use it. First, register windows only help up to a point — the number of physical registers is finite and eventually SPARC runs out of space for more windows. When that happens, you’re back to pushing and popping operands on and off the stack. It’s next to impossible to predict when the register file will overflow or underflow, so performance can be unpredictable. Finally, the processor doesn’t handle the overflow/underflow automatically in hardware. It generates a software fault, which the operating system has to handle, consuming more cycles.

Many hardware engineers aren’t particular fans of register windowing. It puts enormous demands on multiplexers and register ports to make any physical register appear to be any logical register. In the 1990’s when there were nearly a dozen different vendors designing and marketing SPARC-compatible processor, their designers complained bitterly about the headaches in routing interconnect over, around, and through the register file in the middle of every SPARC processor.

That register windowing, which is an inherent and permanent feature of every SPARC, has so far made it impossible to add multithreading, and difficult to keep clock speeds up. The 900-MHz and new 1.05-GHz UltraSPARC-III chips both use TI’s 0.15-micron copper process for their 29 million transistors. Fortunately, most SPARC processors are buried inside Sun workstations, where the value of Sun’s software base and systems-level expertise outshine the relative shortcomings of its processors.

#### 13.5.2 Massive Pipelining

A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different number of steps, they are basically variations of the five steps as follows.

can spend quite a bit of time stalling: waiting for the result of one instruction before it can proceed with a subsequent instruction.

(ii) **Scheduling:** Since the scheduling rules can be complicated, most programmers use a high level language (such as C or C++) and leave the instruction scheduling to the compiler. This makes the performance of RISC application depend critically on the quality of the code generated by the compiler. Therefore, developers (and development tool suppliers such as Apple) have to choose the compiler carefully based on the quality of the generated code.

(iii) **Debugging:** Unfortunately, instruction scheduling can make debugging difficult. If scheduling (and other optimizations) are turned off, the machine-language instructions show a clear connection with their corresponding lines of source. However, once the instruction scheduling is turned on, the machine language instructions for one line of source may appear in the middle of the instructions for another line of source code.

Such an intermingling of machine language instructions not only makes the code hard to read, it can also defeat the purpose of using a source-level complier, since single lines of code can no longer be executed by themselves.

Therefore, many RISC programmers debug their code in an un-optimized, un-scheduled form and then turn on the scheduler (and other optimizations) and hope that the program continues to work in the same way.

(iv) **Code expansion:** Since CISC machines perform complex actions using a single instruction, whereas RISC machines may require multiple instructions for the same action, code expansion can be a problem. Code expansion refers to the increase in size which results from a program that had been compiled for a CISC machine.

Fortunately for us, the code expansion between a 68K processor used in the non-PowerPC Mac and the PowerPC seems to be only 30-50% on the average, although size-optimized PowerPC code can be of the same size (or smaller) than corresponding 68K code.

(v) **On chip cache:** RISC machines require very fast memory systems to feed them instructions. RISC-based systems typically contain large caches, usually on the chip itself.

### 13.7 ARCHITECTURE OF SOME RISC PROCESSORS

In this section we will review some of the popular RISC processors. We will discuss about MIPS, an early RISC processor, followed by SUN UltraSpark.

#### 13.7.1 MIPS

MIPS or "Microprocessor without Interlocked Pipeline Stages" is a RISC architecture without (ideally) any hardware interlocks, a design goal that MIPS has very nearly kept for all these years.

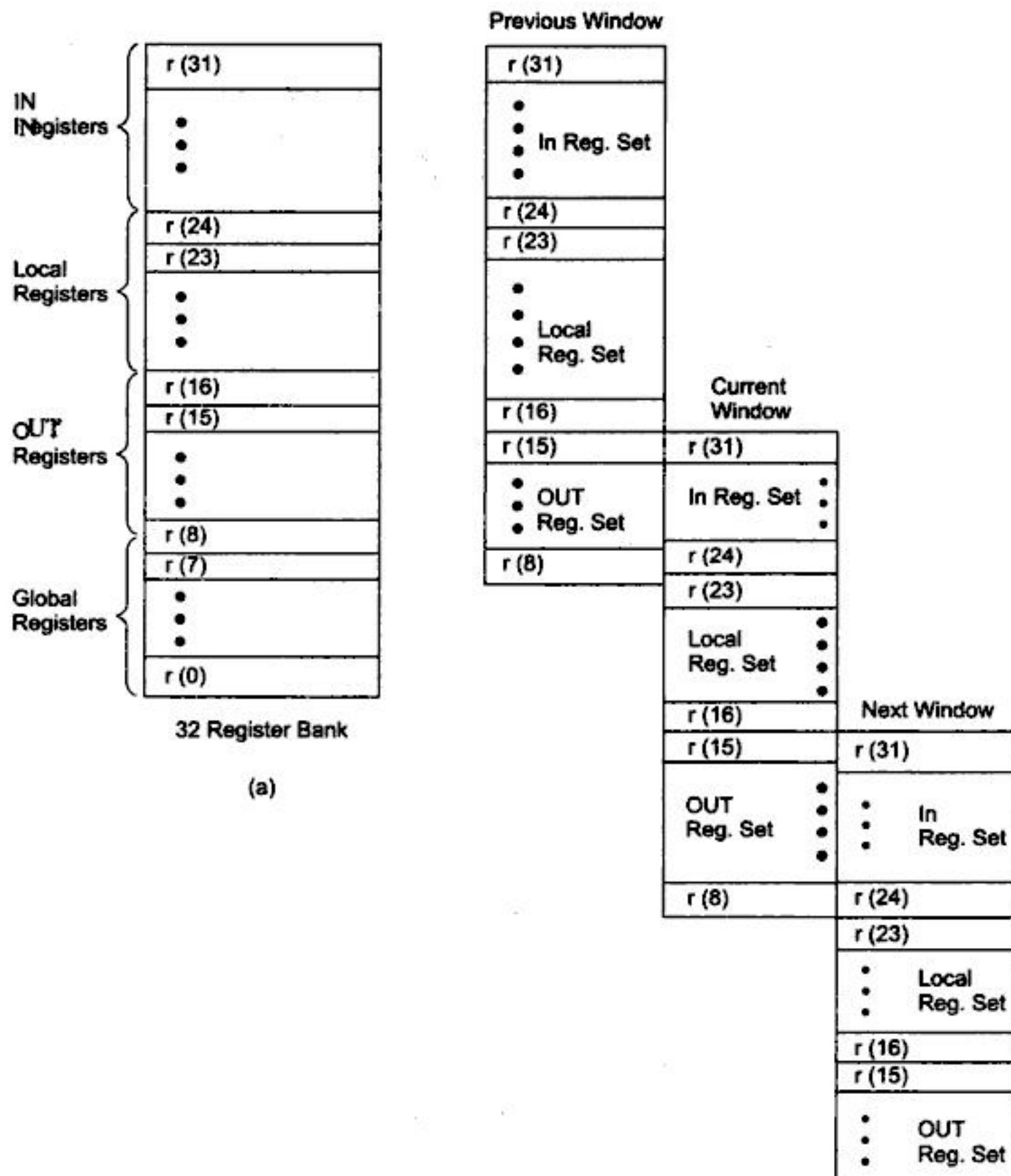
One of the earliest RISC processors was initiated in early 80's at Stanford by Hennessy who subsequently formed in 1984 MIPS Computer systems which brought out 32 bit R2000 in 1985, followed by R3000 in 1988 and 64 bit R4000 in 1991. Subsequently in 1999, 32 bit MIPS 2 and 64 bits MIPS 64 were released.

The MIPS processor used 32 registers, each 32 bits wide. The instruction set consisted of 111 instructions. The instruction set consisted of a variety of basic instructions enlisted as follows:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

parameters may be stored in the register window, without using stacks at all. Also in case of interrupts including TRAP, these register windows are used for saving the context of the processor. In other words the register windows may act as stacks. These register windows may sometimes overlap. This means that there may be some registers which are common to both windows. SPARC processor uses a rotation scheme to ensure the effective allotment of the register windows.



**Fig.13.2 (a) Bank of 32 Registers and their Classification (b) Three Overlapping Register Window in SPARC Architecture.**



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# Microprocessor Based Aluminium Smelter Control

## 14.1 GENERAL PROCESS DESCRIPTION OF AN ALUMINIUM SMELTER

The process of extracting aluminium in a smelter involves electrolysis of  $\text{Al}_2\text{O}_3$  (alumina) in fused cryolite. The electrolysis process takes place in an electrolytic cell, popularly known as a 'Pot'. The electrolytic cell is an iron box internally coated with refractory material. A layer of carbon on this lining acts as cathode of the electrolytic chamber. The anode is made up of an array of carbon rods which can be moved in the vertical direction. Aluminium is produced by electrolytic reduction of alumina ( $\text{Al}_2\text{O}_3$ ) in the electrolytic chamber where the carbon anode and carbon cathode reacts with  $\text{O}_2$  to yield  $\text{CO}_2$  and CO. The configuration of a single electrolytic cell is shown in Fig. 14.1. After aluminium is deposited at the bottom of a bath, it is tapped out from time to time. Only one such electrolytic cell produces very little aluminium and hence a large number of such cells are connected in series, so as to be fed from a single supply, as shown in Fig. 14.2. Each electrolytic cell is connected in such a way that the cathode of the  $i^{\text{th}}$  cell is connected to the anode of the  $(i+1)^{\text{th}}$  cell. The carbon anodes, in the form of carbon rods are dipped inside the electrolyte so that the current needed for electrolysis flows through these anodes in a distributed fashion as shown in Fig. 14.1. Gradually, as the process of electrolysis continues, the carbon gets corroded, and anodes are lowered more and more so that the carbon anodes get dipped into the electrolyte.

In case of a healthy electrolytic cell, the anode to cathode voltage may remain within 4 V to 6 V while the line current may vary between 30 KA to 70 KA. In the next section, we describe the normal control of the electrolytic cell.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

#### 14.4 BRIEF DESCRIPTION OF THE CONTROL LAWS FOR ABNORMAL CELLS

This process controller continuously monitors the system variables like the line current, anode to cathode voltage of each cell, then detects any kind of abnormality in a cell and finally controls the anode position according to the status of the cell.

The control action is initiated for each type of abnormality and continues for a specified period of time depending upon the intensity and type of cell abnormality till the cell becomes normal.

For example, a cell is detected as shaky after monitoring a set of say, 8 or 16 sampled values of anode to cathode voltage and the line current. A cell is considered shaky only if there exists a variation of the measured anode to cathode voltage in the consecutive sampled voltage values for that cell and if this variation goes beyond a threshold voltage, say 1V.

For removing of shakiness, the target voltage  $V_t$  is chosen as:

$$V_t = V_{t\text{ basic}} + V_{t\text{shaky}} \quad (14.3)$$

Where  $V_{t\text{ basic}}$  is the basic target anode to cathode voltage for a cell and is to be specified by the control operator and  $V_{t\text{shaky}}$  is the extra target voltage to be considered only for a shaky cell.

The anode raising and lowering control continues considering  $V_t$  as the total target voltage and the process is continued for hours until the cell behaves in a normal way.

In case of a shaky cell, the extra target voltage  $V_{t\text{shaky}}$  which is provided for computing  $V_t$ , should be continued in steps for a specified time duration, say,  $T_{\text{shaky}}$  (in hours) and the excess target voltage should also be withdrawn in steps of  $T_{\text{shaky}}$  (in hours) time.

As in the case of a shaky cell, the rodded and tapped cells also receive extra target voltage  $V_{t\text{rod}}$  and  $V_{t\text{tap}}$  which is increased in steps of  $T_{\text{rod}}$  and  $T_{\text{tap}}$  time duration. Also the excess target voltage applied is reduced in steps of  $T_{\text{rod}}$  and  $T_{\text{tap}}$  time duration, till the cell comes back to the normal condition. The most important type of control is, however, the anode effect quenching program which is explained next.

**Anode Effect Quenching** A cell is considered to be in the anode effect, if the anode to cathode voltage  $V_m$  is found to exceed a certain threshold limit, say 10V to 15V, depending upon the nature of the cell. During anode effect, sometimes the measured voltage,  $V_m$ , may shoot up to a large value, and thus the line current,  $I_m$ , may be reduced. Moreover, if a number of cells are affected with the anode effect, there may be a considerable drop in the line current. Thus this drop in the measured line current may be indicative of the anode effect in one or more cells in the smelter. A quick scan of the anode to cathode voltage of all the cells in the smelter network may reveal the status of the cell, i.e. whether it has been affected by anode effect or not.

In case, anode effect is detected in a cell, the cell is immediately isolated and anode effect quenching program is executed. This involves lowering of the anode in steps till the anode moves very close to the cathode, resulting in a spark which breaks the crust and cleans the cell. There is a pause between every lowering operation resulting in a pattern—lower ( $T_1$ ), pause ( $T_2$ ), lower ( $T_1$ ), pause ( $T_2$ ) ... for specified durations,  $T_1$  and  $T_2$  respectively. Likewise, the anode is next raised in steps with a pause between every raise operation. This full cycle of lowering and raising of the anodes stepwise is continued for a large number of cycles, till the cell becomes normal.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

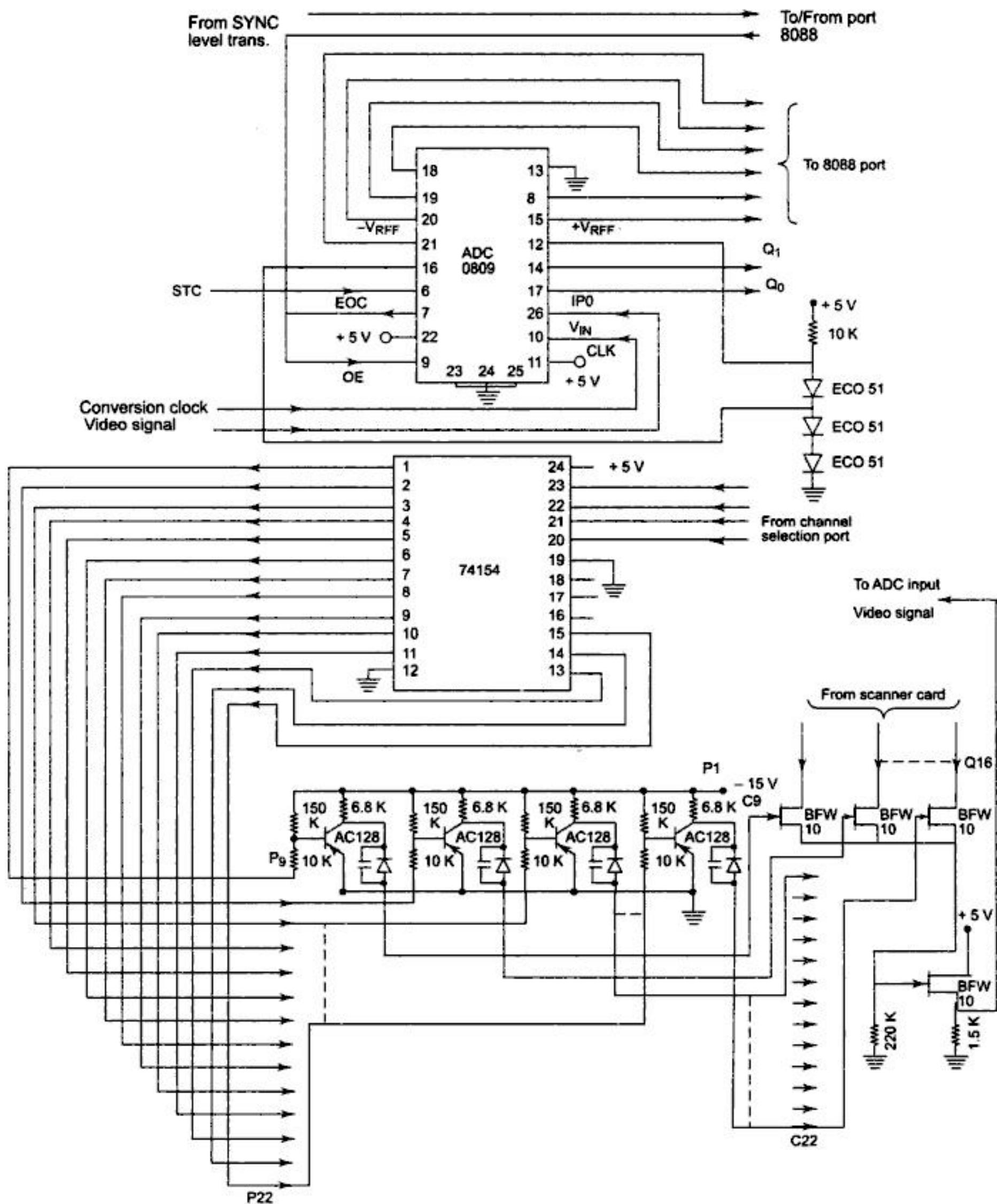


Fig. 15.4 Converter and Multiplexer Interface



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

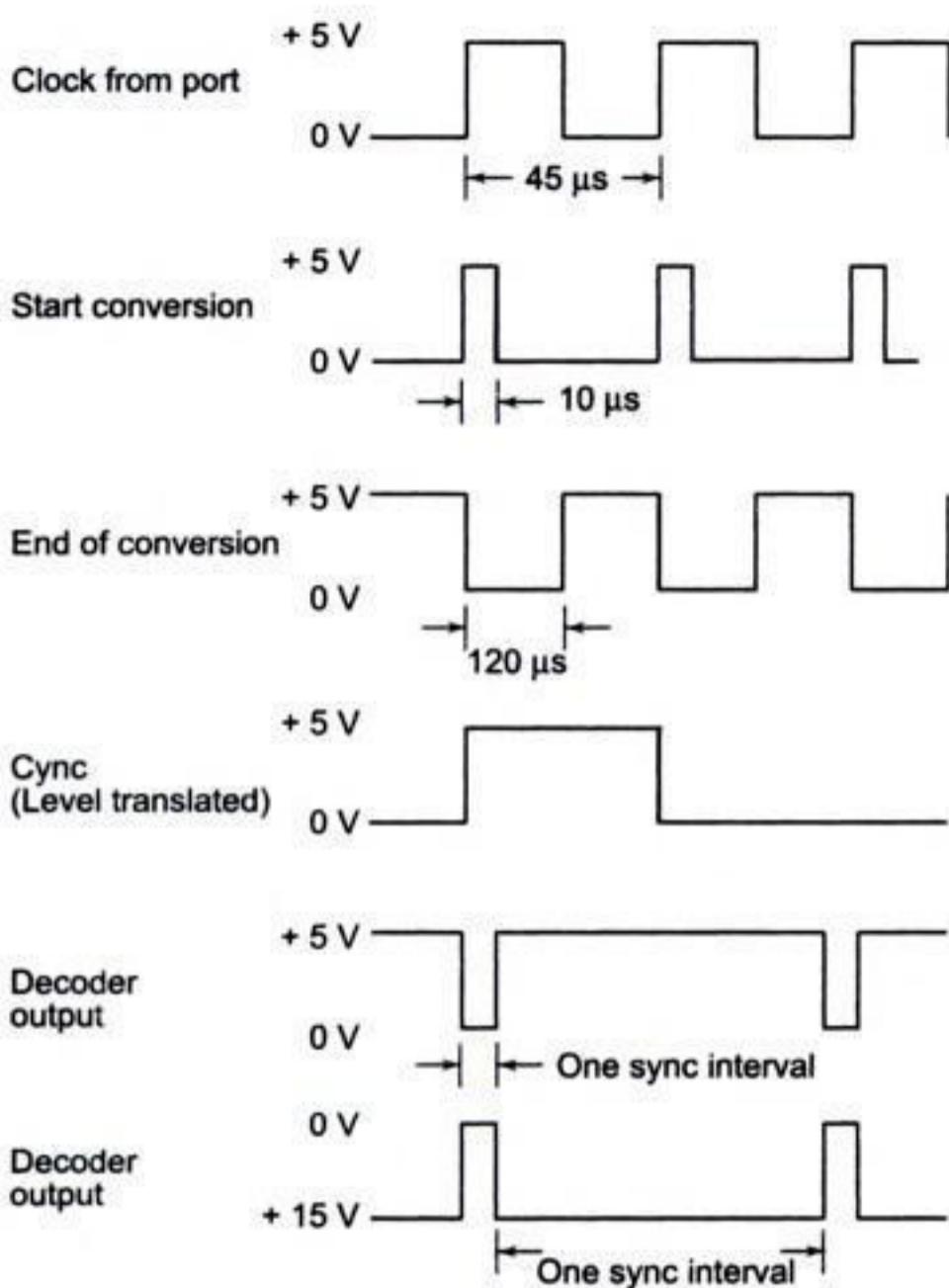


Fig. 15.8 Timing Diagram

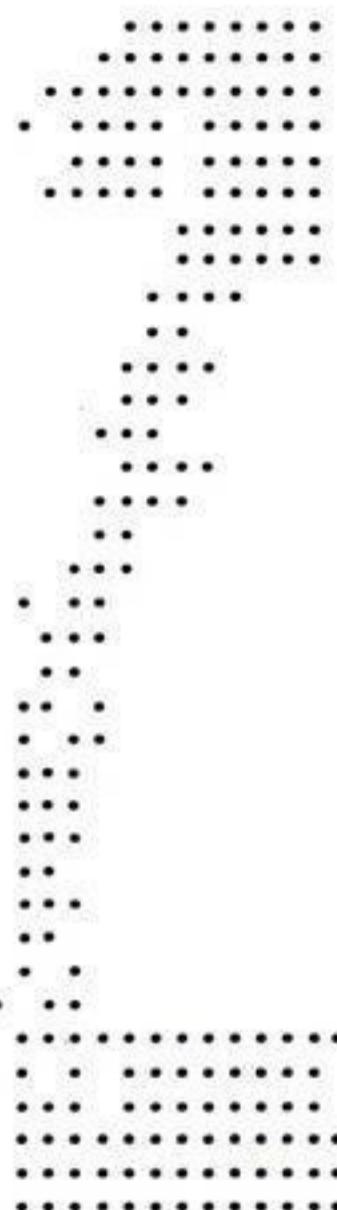


Fig. 15.9 Digitized Stored Pattern of Numeral '2'

## SUMMARY

The program-controlled data acquisition system is very reliable and of higher precision than systems using other modes of storage. In the first two options of storage mode, as mentioned earlier, a Schmitt trigger has been used with typical threshold levels 1.7 V (positive going) and 0.9 V (negative going). The precision is obtained in program-controlled mode by the use of an A/D converter. However, the conversion time adds to the loop execution time for the clock generation resulting in a decrease of storage speed. The local storage mode is thus faster than the program-controlled mode. 256 grey levels may be obtained between the dark and saturation levels.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



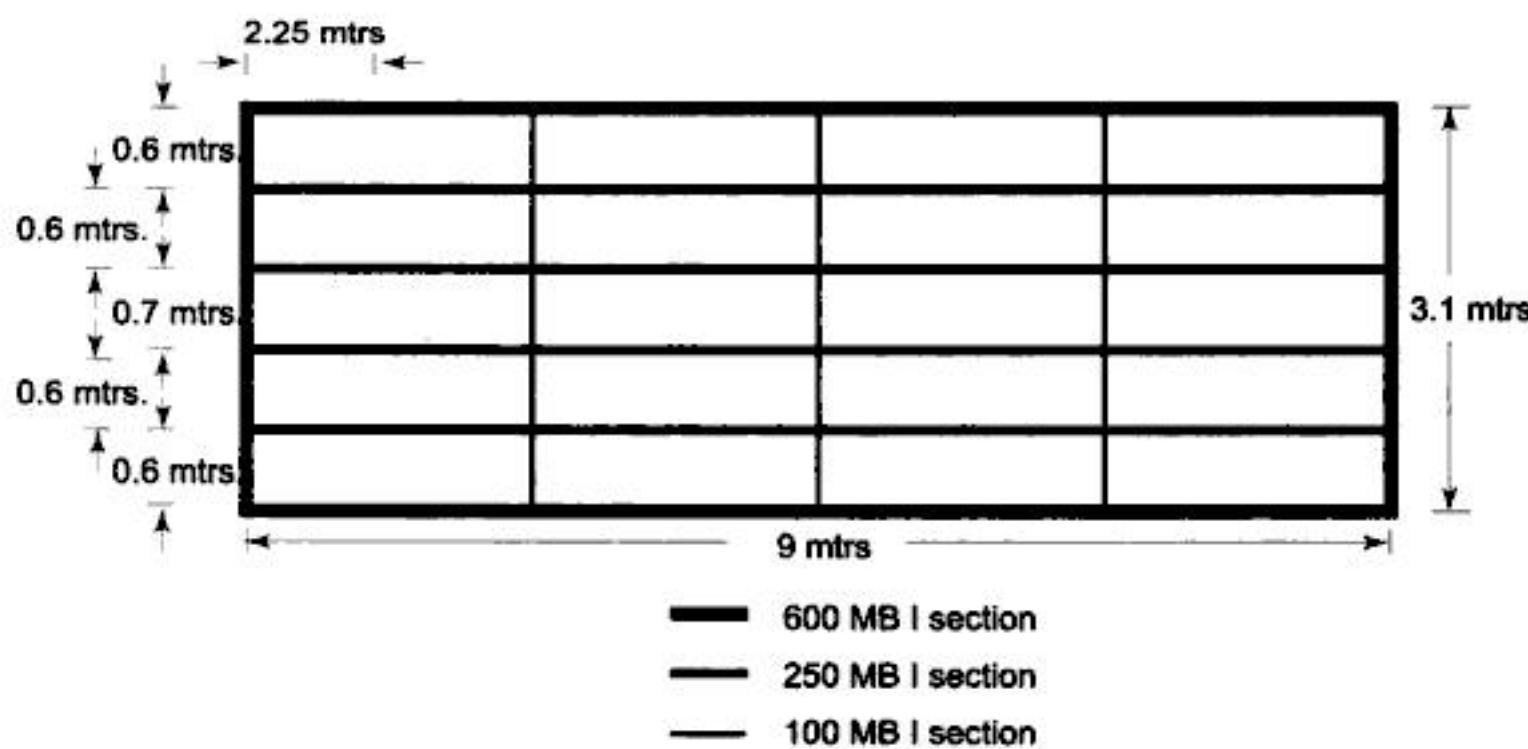
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

without any permanent deformation or damage in it. If at all any deformation of the structure takes place, it must be within the elastic limits of the material and the platform. A safety factor of 3 is considered while designing the mechanical structure. The mechanical platform rests over the load cells, which are physically quite small as compared to the platform, and thus the platform may topple down the load cells and may get dislodged. Hence the mechanical structure should have some fixing arrangements (or separate fixtures and mountings may be designed) for fixing the mechanical structure with the load cells. The mechanical structure such mounted should have a sufficient clearance between it and the road ramp, to float the structure laterally between the two ramps. This lateral movement of the structure and the load cell fixtures and mountings prevent the load cells from toppling, due to the lateral forces appearing due to vehicle movement and abrupt braking. The complete mechanical structure must be coplanar to avoid corner error. A typical mechanical structure for a 40 tonnes weigh-bridge is 9 meters in length and 3.1 meters in breadth to carry a full normal size truck vehicle. A number of mechanical structure designs may be suggested, using structural engineering principles to serve the purpose. A simple mechanical structure for a 40 tonnes weigh-bridge is shown in Fig. 16.2.

The mechanical structure should have pipes attached to it along its edges for carrying the load cell wires to the adder circuit, for adding all the load cell signals. The adder circuit is located in a cabinet usually attached with the mechanical structure or mounted near by. The output of the adder circuit is further amplified by the signal processing circuit. During the weighing procedure, the mechanical structure should rest only on the load cells. It should neither touch the road ramp nor any additional support except the load cell fixtures.

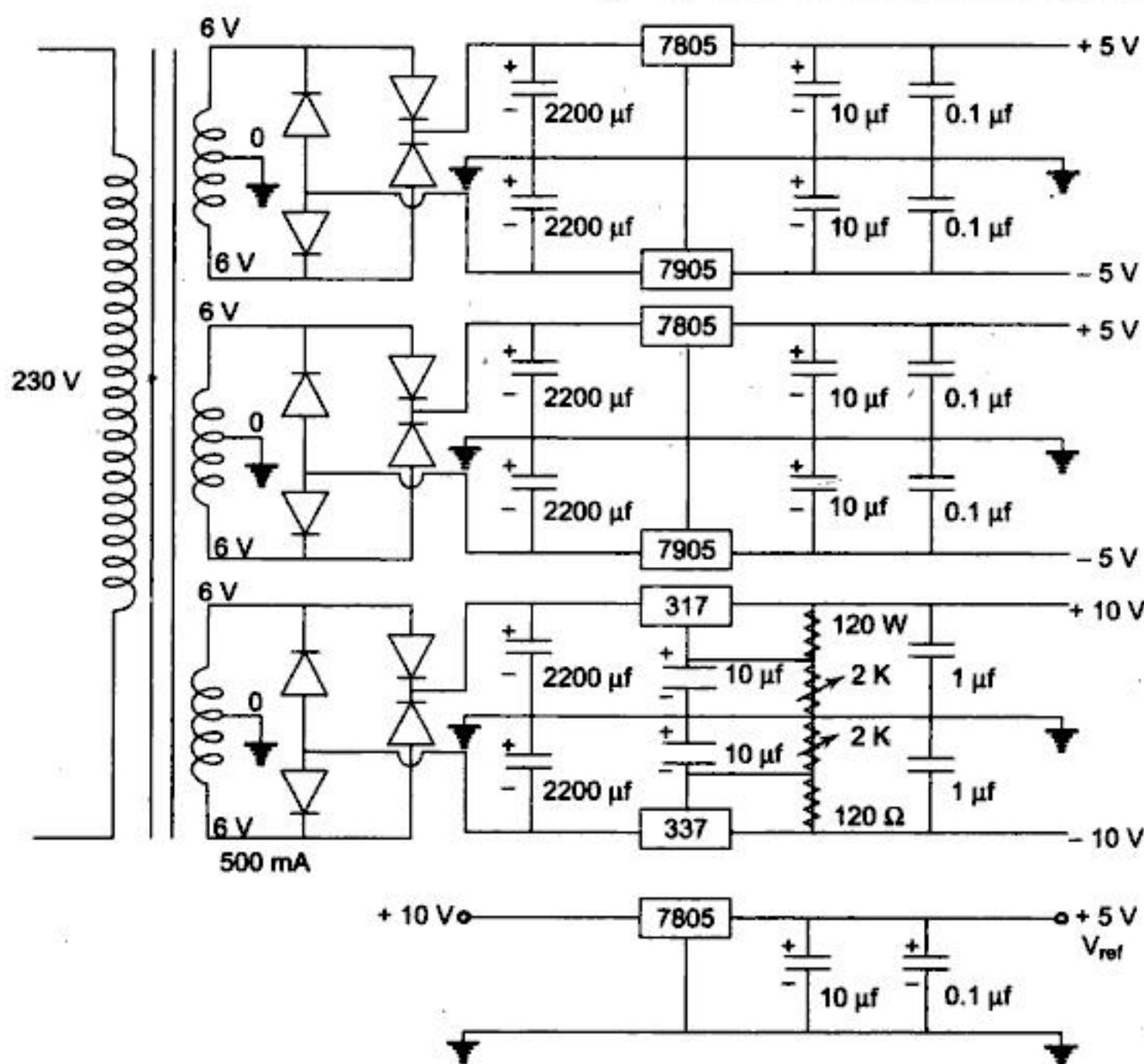


**Fig. 16.2 Mechanical Structure**

### 16.1.3 Load Cells-Selection and Connections

In the most popular form, a load cell contains a mechanical structure machined out of suitable types of steels or alloys and heat treated to maintain the material characteristics for a long period. This mechanical structure acts as a primary transducer to convert the weight to which it is subjected into its proportional strain. This strain is measured using strain gauges which are usually connected in the form of wheatstone

+5 V and -5 V supplies. The +10 V and -10 V supply may require fine adjustments at the time of calibration. Hence LM317 and LM337 variable voltage regulators are used for deriving these supplies.



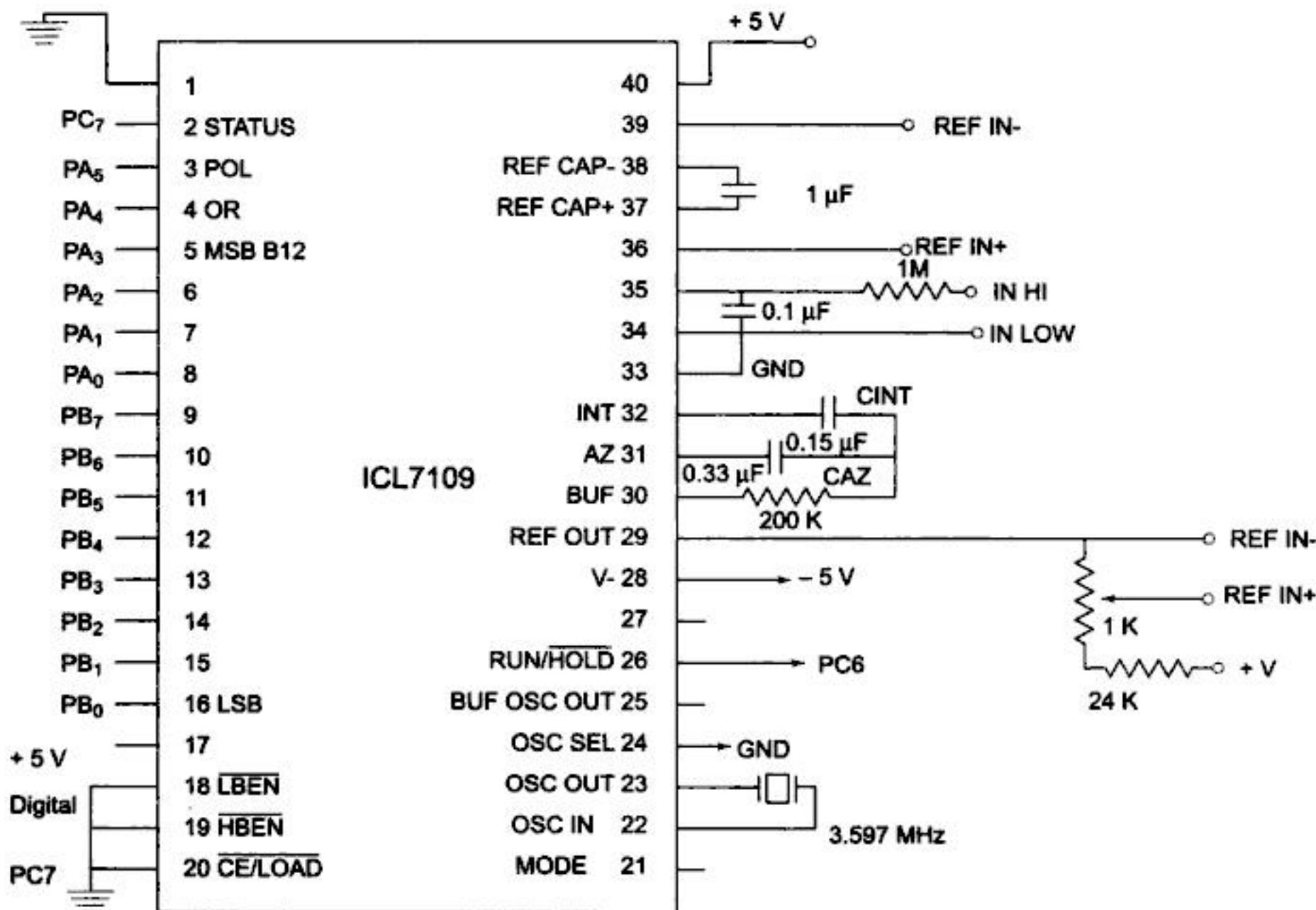
**Fig. 16.6 Power Supply Circuit**

### Signal Processing Circuit

**Signal Adder Circuit (Adder/Summer)** The adder circuit accepts individual outputs of the load cells and derives a common output signal which is further fed to the instrumentation amplifier. The adder circuit has already been shown in Fig. 16.4. The four bridges in Fig. 16.4 represent four load cells. The resistances  $R_1$ ,  $R_2$ ,  $R_3$  and  $R_4$  are used for minute corner adjustments. The resistance  $R_5$  is used for zero load adjustment (offset adjustment) of the system.

**Instrumentation Amplifier** This circuit is the most important and sensitive part of the system which amplifies a signal derived by the adder circuit. A number of single chip instrumentation amplifiers are available commercially. A few of the popular ones are AD625, LM363 and ICL7650, etc. Most of the single chip instrumentation amplifiers are costly components. Hence a comparatively simple and cheaper

get a stable weight display. To overcome this problem average or moving average of samples may be computed and used as an equivalent digital count, to get a stable display. The ADC circuit is shown in Fig. 16.8. The output of the ADC circuit is read by the CPU using an I/O port. The ADC circuit requires a proper software support to function properly. The flowchart and program for ADC operation has already been discussed in Chapter 5.



**Fig. 16.8** Analog to Digital Converter

**Microprocessor System** The microprocessor system designed for this application is shown in Fig. 16.9. The system is designed around 8088, which offers the simplicity of 8-bit processors for peripheral interfacing and the powerful instruction set of the 16-bit processor 8086. The connector J<sub>1</sub> is a power supply connector. The connector J<sub>2</sub> makes the CPU system bus interrupts and control bus available for external connections. The system is able to address 32 Kbytes of EPROM and 32 Kbytes of RAM available on the board. An additional memory bank of 960 Kbytes, may be designed using thirty 62256 RAM ICs and can be readily connected at socket J<sub>3</sub>, which provides the chip select signals for the external 62256 RAM ICs. The chip selects of these additional RAMs are readily designed on the board using two 74154 and are brought out at connector J<sub>3</sub>, along with power supply pins.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

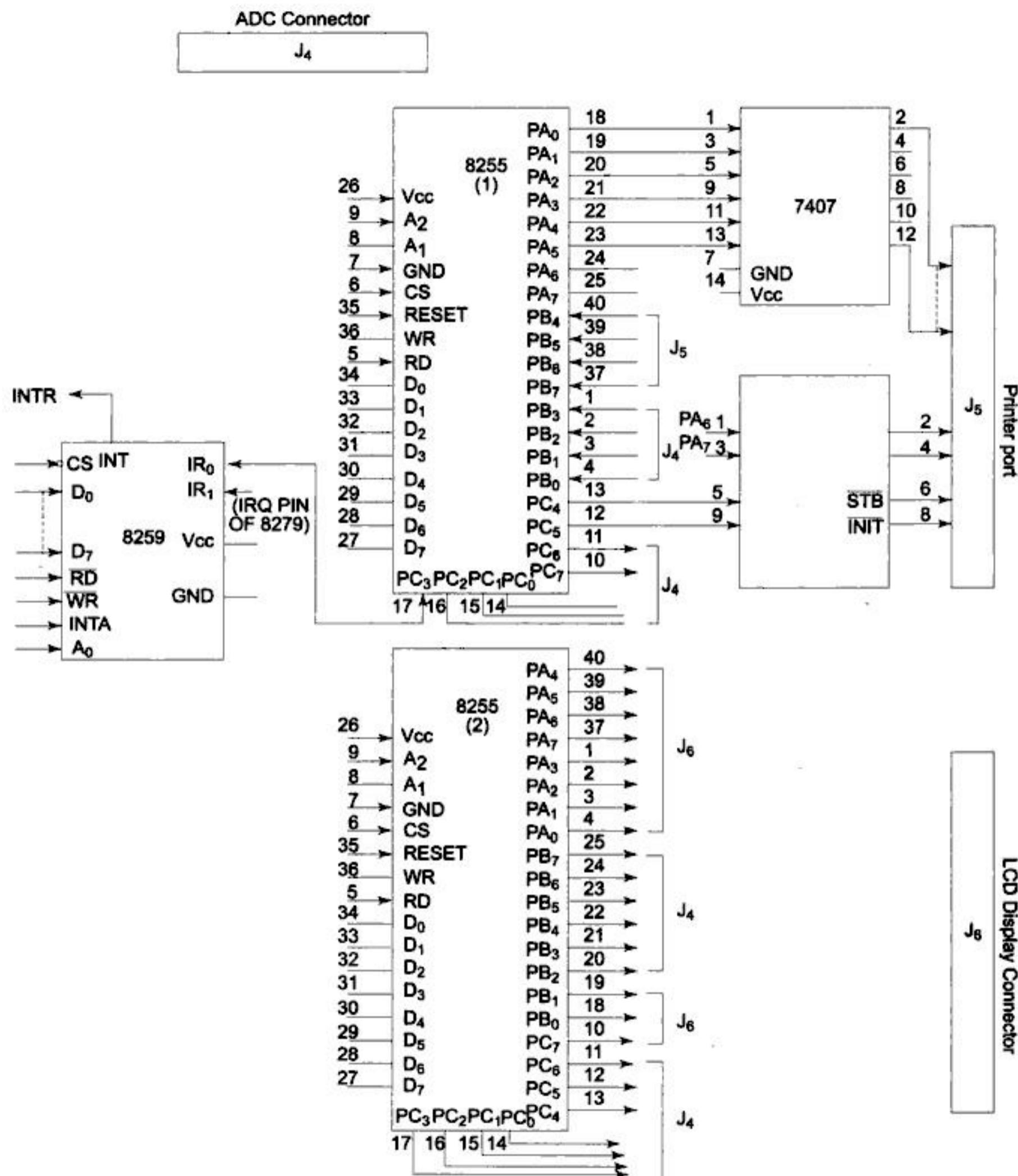


Fig. 16.9 Microprocessor 8088 Based System Circuit Diagram for the Electronic Weighing Bridge

The connector J<sub>4</sub> has the serial communication signals for RS 232C. The RS 232 compatible line drivers are used for their usual purpose along with 8251 for serial communication with a PC. The connector J<sub>5</sub> is a 25-pin connector at which a printer may be connected. The printer is interfaced using 8255 as already discussed in Chapter 5. The connector J<sub>6</sub> is a 26-pin 8255 connector and may be used for interfacing a LCD alphanumeric display and driver module, for weighing ticket data entry, like seller's name, purchaser's name, tare weight, etc. The J<sub>6</sub> also interfaces a 12-bit ADC with the CPU. Note that when the data entry on the LCD module is in process the ADC is hold. The connector J<sub>7</sub> presents the signals required for keyboard and display interfacing made available by the on-board 8279.

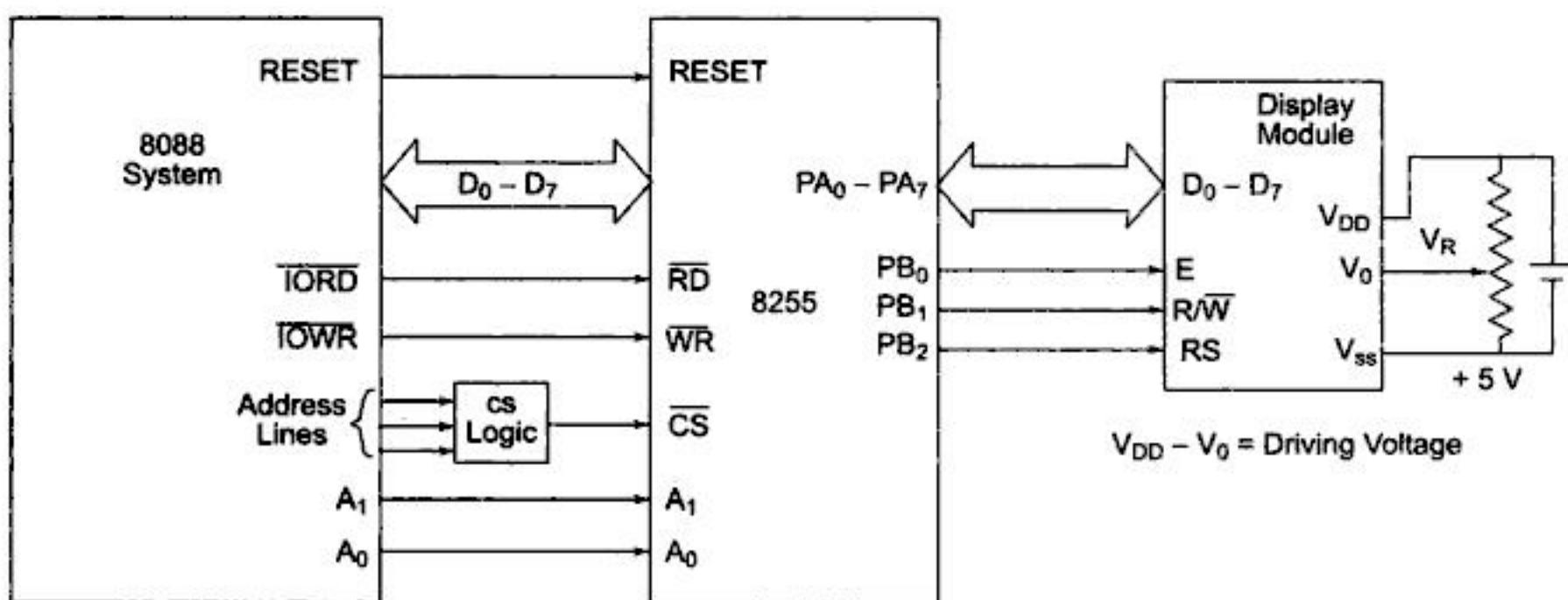
The microprocessor system contains on-board 32 Kbytes of RAM, 32 Kbytes of EPROM, 8279 keyboard display controller, 8253 programmable timer, 8251 serial communication interface and two 8255's out of which, the first is for a printer interface and the other for a LCD display driver interface. Interfacing techniques of all these peripherals have already been discussed in details in this text except for a LCD driver module.

The latches 74373 and buffers 74245 have been used for latching the addresses and separating the data from the multiplexed address/data signals generated by the CPU. The decoder 74154 is used for generating the chip selects of the 32 Kbytes EPROM and RAM memory chips (either on-board or external). A 3:8 decoder 74138 is used to generate the chip select lines of all the on-board peripheral circuits. Another 74138 is used for decoding the keyboard/display scan lines from the encoded scan output lines generated by 8279. The line receiver 1489 and the line driver 1488 are used for interfacing the 8251 signals with the RS-232 communication standard. The buffers 7407 are used for interfacing the printer data bus and control signals with 8255 I/O lines. Figure 16.9 shows the complete microprocessor system circuit diagram.

The 64-key keyboard is shown along with the key assignments in Fig. 16.10. The keyboard and the five unit 7-segment display are to be connected with the 8279 at connector J<sub>7</sub> using a flat crimped connector. Figure 16.11 shows the 7-segment display unit.

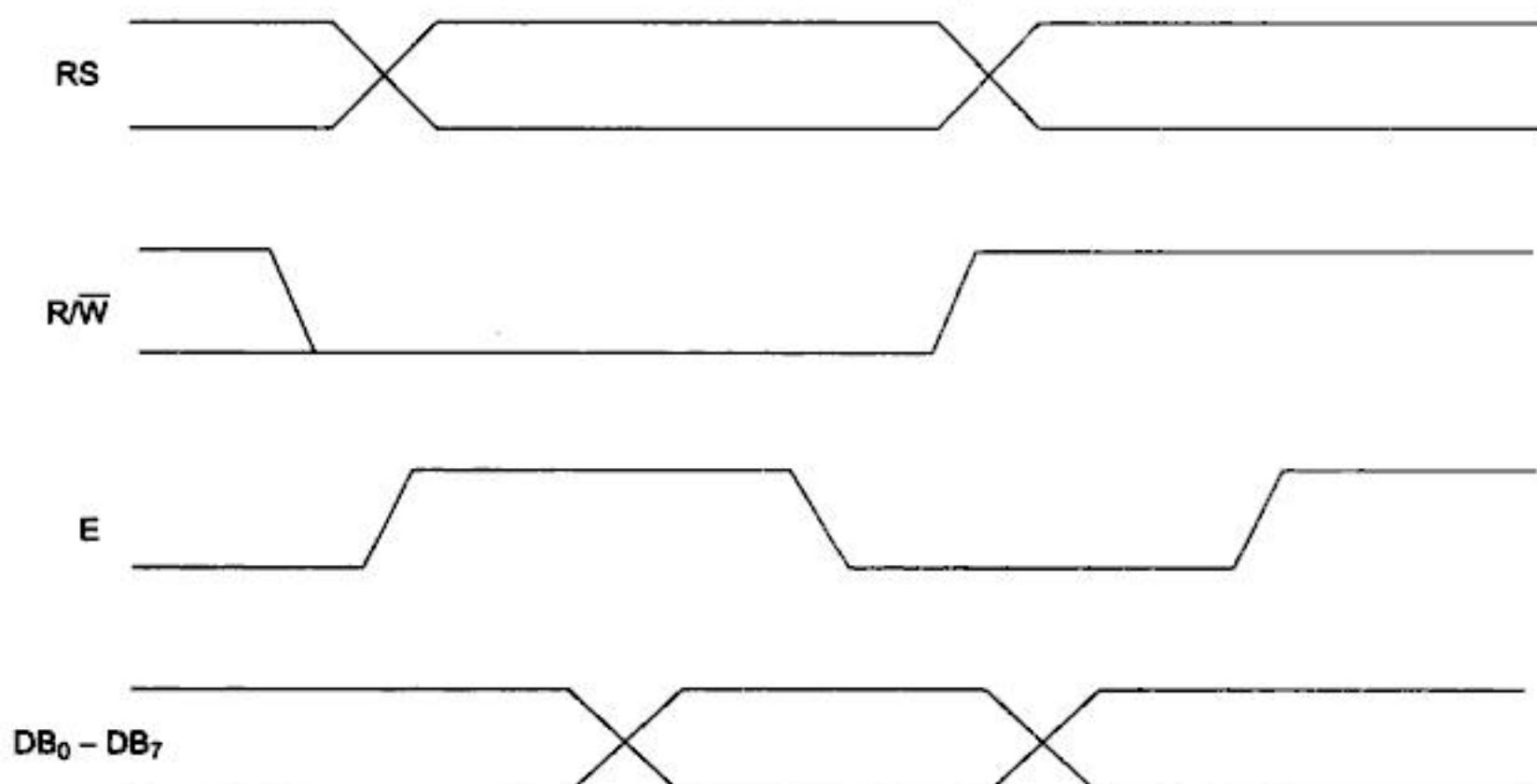
**Interfacing LCD Module 'ORIOLE'** The system provides a LCD dotmatrix 80-character display, organised as two lines, each of 40 characters. A custom built module containing this display with its controller is readily available in the market. Here we have used one such LCD module from 'ORIOLE'. This module can be interfaced with a CPU using I/O ports. Here we have used programmable I/O ports of 8255 for interfacing the LCD module with 8088. The LCD module is to be connected at socket J<sub>6</sub>. Figure 16.12 shows the interfacing connections of 8088 with the LCD module using 8255.

The display module has an  $80 \times 1$  byte display buffer with two internal registers namely, instruction and data registers. The eight data lines D<sub>0</sub>-D<sub>7</sub>, carry data/control words from/to the CPU to/from the display module. The E (Enable) input line, when high, enables a read/write operation from/to the LCD controller. Once an instruction is written into the instruction register, the internal BUSY status goes high. No further operation is possible with the LCD controller till the BUSY flag is high, i.e. the controller is busy. After the specified busy time, the contents of the address counter that contain the current display pointer can be read by the CPU. The busy status flag can be continuously read, while the controller is in busy status, and whenever it shows NO BUSY status, further address counter read operation can be carried out. The busy state is presented by the D7 line when the E line goes high. The R/W line indicates whether it is a read or a write operation. For the write operation, the R/W line should be low while for the read operation it should be high. The write operation issues an instruction

**Fig. 16.12** Interfacing LCD Module with 8088

or a data byte to the controller, while the read operation either reads the busy status or the address counter content. The RS (Register Select line) selects one of the two register, viz. instruction register and data register for the selected read/write operation.

The write and read operations of the LCD controller are presented in Fig. 16.13 and Fig. 16.14 along with the critical timings. The specified timings for the respective operations should be strictly followed, failing which the LCD module and PIO may have permanent damage.

**Fig. 16.13** Write to LCD Controller Timings



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

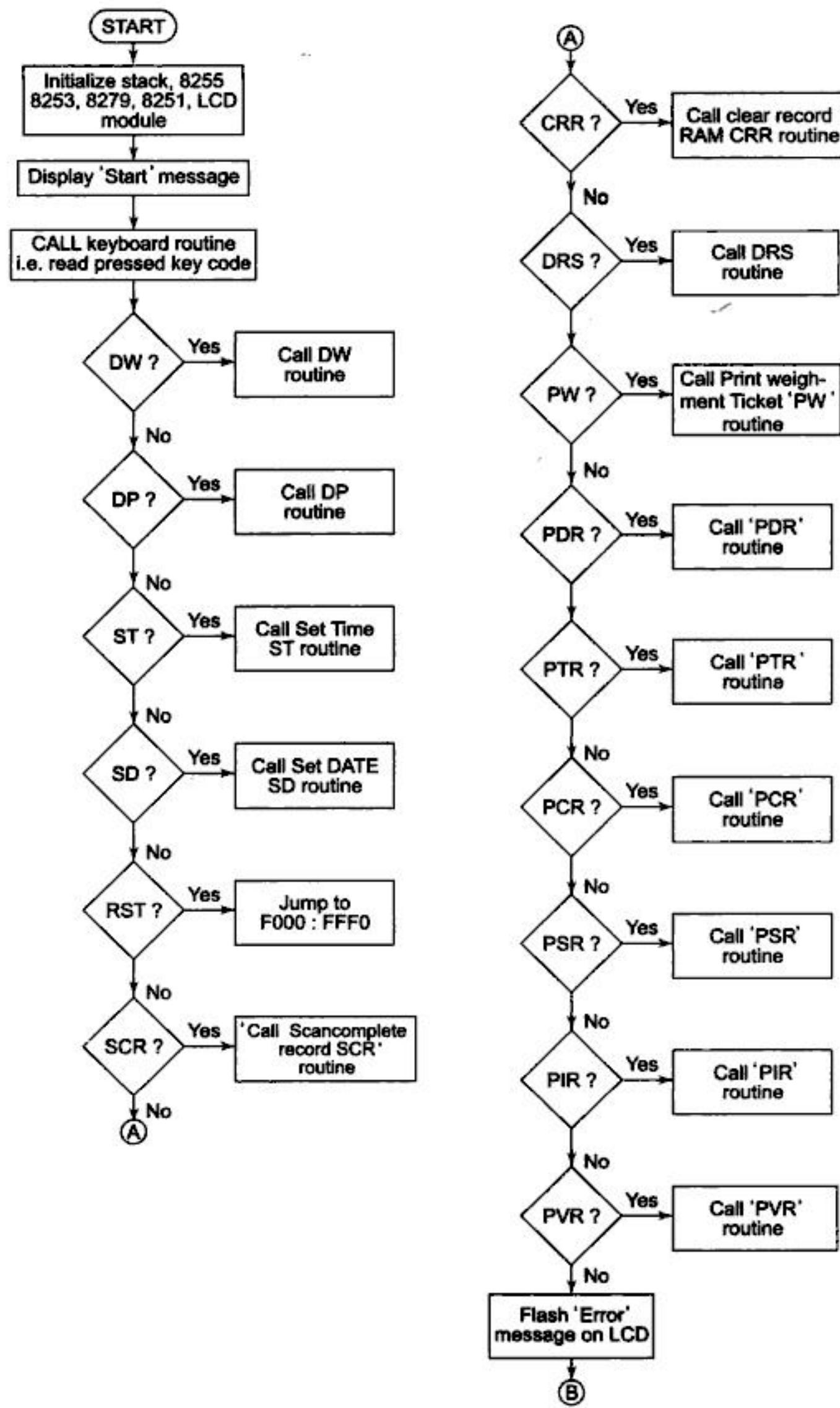


Fig. 16.16 Complete Flow Chart of the Main Program



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

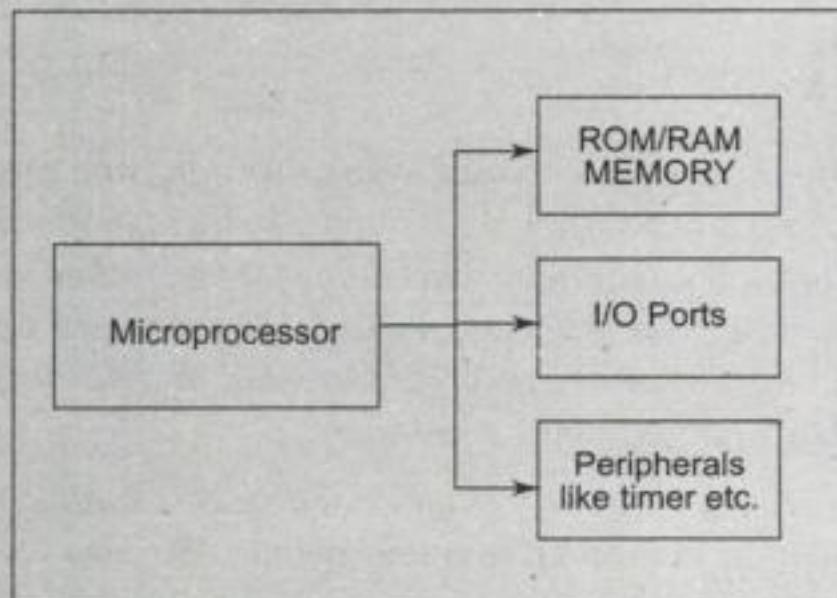


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Considering all these problems, Intel decided to integrate a microprocessor along with I/O ports and minimum memory into a single package. Another frequently used peripheral, a programmable timer, was also integrated to make this device a self-sufficient one. This device which contains a microprocessor and the above mentioned components has been named a microcontroller. A microcontroller a microprocessor with integrated peripherals. The introduction of microcontrollers drastically changed the microprocessor based system design concepts, specially in case of small dedicated systems. Design with microcontrollers has the following advantages:

1. As the peripherals are integrated into a single chip, the overall system cost is very low.
2. The size of the product is small as compared to the microprocessor based systems thus very handy.
3. The system design requires very little efforts and is easy to troubleshoot and maintain.
4. As the peripherals are integrated with a microprocessor, the system is more reliable.
5. Though a microcontroller may have on-chip RAM, ROM and I/O ports, additional RAM, ROM and I/O ports may be interfaced externally, if required.
6. The microcontrollers with on-chip ROM provide a software security feature which is not available with microprocessor based systems using ROM/EPROM.
7. All these features are available in a 40 pin package as in an 8-bit processor.

However, in case of a larger system design, which requires more number of I/O ports and more memory capacity, the system designer may interface external I/O ports and memory with the system. In such cases, the microcontroller based systems are not so attractive as they are in case of the small dedicated systems. Figure 17.1 shows a typical microcontroller internal block diagram.



**Fig. 17.1 Microcontroller Internal Block Diagram**

As a microcontroller contains most of the components required to form a microprocessor system, it is sometimes called a single chip microcomputer. Since it also has the ability to easily implement simple control functions, it is most frequently called a microcontroller.

In this chapter, we will briefly present Intel's 8-bit microcontroller family, popularly known as MCS-51 family. In the end, we will introduce the architecture and general features of Intel's 16-bit microcontroller family called MCS-96.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

