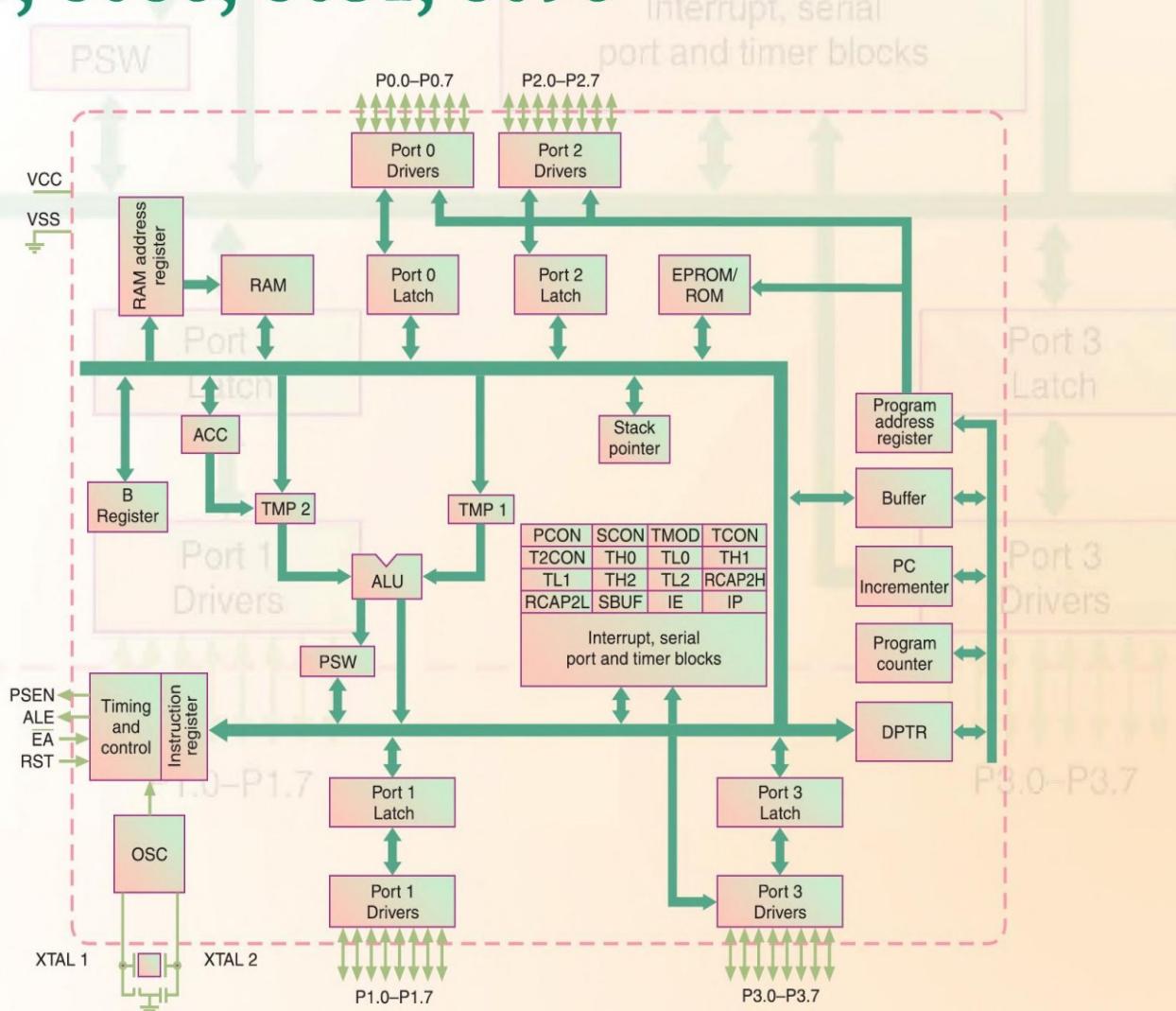


Second Edition

Eastern
Economy
Edition

MICROPROCESSORS AND MICROCONTROLLERS

Architecture, Programming
and System Design
8085, 8086, 8051, 8096



Krishna Kant

PHI

MICROPROCESSORS AND MICROCONTROLLERS

**Architecture, Programming and System
Design
8085, 8086, 8051, 8096**

Krishna Kant
Dean (Academic)
Jaypee Institute of Information Technology
Noida

PHI Learning Private Limited
Delhi-110092
2012

**MICROPROCESSORS AND MICROCONTROLLERS:
Architecture, Programming and System Design 8085, 8086, 8051, 8096**
Krishna Kant

© 2007 by PHI Learning Private Limited, Delhi. All rights reserved. No part of this book may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publisher.

ISBN-978-81-203-3191-4

The export rights of this book are vested solely with the publisher.

**Ninth Printing
2012**

...

...

July,

Published by Asoke K. Ghosh, PHI Learning Private Limited, 111, Patparganj Industrial Estate, Delhi-110092 and Printed by Rajkamal Electric Press, Plot No. 2, Phase IV, HSIDC, Kundli-131028, Sonepat, Haryana.

To
Solid Pillars of My Life
Wife Dr. Madhu Chhanda
Our Son Prashant Rishi
and Our Daughter Neha Shikha

CONTENTS

<i>Preface</i>	<i>xv</i>
<i>Acknowledgements</i>	<i>xix</i>

1. System Microprocessor Design 1–16 Using

1.1 Introduction	1
1.2 System Design	2
1.2.1 Feasibility Study	2
1.2.2 Random Logic vs. Microprocessor	3
1.2.3 System Specification	5
1.2.4 Initial Design	6
1.2.5 Hardware Design	6
1.2.6 Software Design	6
1.2.7 Test and Debug	11
1.2.8 Integration	11
1.2.9 Documentation	11
1.3 Development Tools	11
1.3.1 Microcomputer Kit	12
1.3.2 Dedicated Microprocessor Development System	12
1.3.3 Universal Microprocessor Development System	15
1.3.4 Simulators	15
1.4 Conclusion	15
<i>Exercises</i>	16
<i>Further Reading</i>	16

2. What a Microprocessor Is 72 17–

2.1 Introduction	17
2.2 Computer and Its Organization	17
2.2.1 Input System	18
2.2.2 Output System	18
2.2.3 Arithmetic and Logic Unit (ALU)	18
2.2.4 Memory	19
2.2.5 Control System	20
2.2.6 Instruction Execution	21
2.2.7 Clock	22
2.2.8 Instruction Format	22
2.2.9 Addressing Modes	23
2.2.10 Instruction Set	27
2.3 Programming System	29
2.3.1 Machine Language Program	29
2.3.2 Assembly Language Program	31
2.3.3 Assembler Directives	31
2.3.4 Compilers	33
2.3.5 Operating Systems	34

2.4 What is Microprocessor?	34
2.5 Address Bus, Data Bus, and Control Bus	35
2.6 Tristate Bus	36
2.7 Clock Generation	37
2.8 Connecting Microprocessor to I/O Devices	38
2.8.1 I/O Mapped I/O Interface	38
2.8.2 Memory Mapped I/O Interface	39
2.9 Data Transfer Schemes	40
2.9.1 Parallel Data Transfer	41
2.9.2 Serial Data Transfer	44
2.10 Architectural Advancements of Microprocessors	45
2.10.1 Pipelining	45
2.10.2 Cache Memory	47
2.10.3 Memory Management	50
2.10.4 Virtual Memory System	51
2.11 Evolution of Microprocessors	54
2.11.1 8-Bit Microprocessors	55
2.11.2 16-Bit Microprocessors	57
2.11.3 32-Bit Microprocessors	61
2.11.4 Bit-slice Processor	65
2.11.5 Microcomputers and Microcontrollers	65
2.11.6 The Transputer	69
2.12 Conclusion	71
<i>Exercises</i>	71
<i>Further Reading</i>	72

3. Intel 8085 Architecture	8085 73–96	Microprocessor—Hardware
3.1 Introduction	73	
3.2 Hardware Architecture	73	
3.2.1 The 8085 Clock	73	
3.2.2 Programmable Registers	75	
3.2.3 Address and Data Buses	76	
3.2.4 Memory Interfacing	77	
3.2.5 Interrupt System	80	
3.2.6 Direct Memory Access	83	
3.2.7 Serial Input–Output	83	
3.2.8 The 8085 Activity Status Information	83	
3.2.9 The 8085 Reset	84	
3.3 The 8085 Pin Out	84	
3.3.1 The 8085 Signals	85	
3.4 Instruction Execution	87	
3.5 Direct Memory Access Timing Diagram	91	
3.6 External Interrupts Timing Diagram	93	
3.7 Conclusion	94	
<i>Exercises</i>	95	
<i>Further Reading</i>	96	

4. Intel 8085 Microprocessor—Instruction Set and Programming	97–126
4.1 Introduction	97
4.2 Program Status Word	98

4.3 Operand Types	99
4.4 Instructions Format	99
4.5 Addressing Modes	99
4.6 Instruction Set	102
4.6.1 Symbols and Abbreviations	102
4.6.2 Data Transfer Instructions	103
<i>Exercises</i>	105
4.6.3 Arithmetic Instructions	106
<i>Exercises</i>	110
4.6.4 Logical Instructions	110
<i>Exercises</i>	115
4.6.5 Branch Instructions	115
<i>Exercises</i>	120
4.6.6 Stack I/O and Machine Control Instructions	121
4.7 Conclusion	124
<i>Exercises</i>	125
<i>Further Reading</i>	126

5.

Intel

8086—Hardware

Architecture	127–184
5.1 Introduction	127
5.2 Architecture	128
5.2.1 Bus Interface Unit (BIU)	129
5.2.2 Execution Unit	129
5.3 Pin Description	136
5.4 External Memory Addressing	142
5.5 Bus Cycles	146
5.5.1 Memory or I/O Read for Minimum Mode	146
5.5.2 Memory or I/O Write for Minimum Mode	148
5.6 Some Important Companion Chips	150
5.6.1 Clock Generator Intel 8284A	150
5.6.2 Bidirectional Bus Transceiver Intel 8286/8287	154
5.6.3 8 Bit Input–Output Port Intel 8282/8283	155
5.6.4 Bus Controller Intel 8288	156
5.7 Maximum Mode Bus Cycle	159
5.7.1 Memory Read Bus Cycle	159
5.7.2 Memory Write Bus Cycle	160
5.8 Intel 8086 System Configurations	161
5.9 Memory Interfacing	164
5.10 Minimum Mode System Configuration	169
5.11 Maximum Mode System Configuration	169
5.12 Interrupt Processing	169
5.12.1 Software Interrupts	173
5.12.2 Single Step Interrupt	174
5.12.3 Non-Maskable Interrupt	175
5.12.4 Maskable Interrupt	176
5.12.5 Interrupt Priorities	177
5.13 Direct Memory Access	177
5.14 Halt State	180
5.15 Wait for Test State	181
5.16 Comparison between the 8086 and the 8088	181

5.17 Compatibility between the 8088, the 8086, the 80186 and the 80286 Processors	182
5.18 Conclusion	182
<i>Exercises</i>	182
<i>Further Reading</i>	184

6. Intel 8086 Microprocessor—Instruction Set and Programming 185–244

6.1 Introduction	185
6.2 Programmer's Model of Intel 8086	185
6.3 Operand Types	187
6.4 Operand Addressing	188
6.4.1 Register Addressing Mode	189
6.4.2 Immediate Addressing Mode	189
6.4.3 Direct Memory Addressing Mode	189
6.4.4 Register Indirect Addressing Mode	191
6.4.5 Base Plus Index Register Addressing Mode	192
6.4.6 Register Relative Addressing Mode	193
6.4.7 Base Plus Index Register Relative Addressing Mode	194
6.4.8 String Addressing Mode	196
6.5 Intel 8086 Assembler Directives	197
6.5.1 Directives for Constant and Variable Definition	197
6.5.2 Program Location Control Directives	198
6.5.3 Segment Declaration Directives	200
6.5.4 Procedure and Macro-related Directives	201
6.5.5 Other Directives	202
6.6 Instruction Set	203
6.7 Data Transfer Group	204
<i>Exercises</i>	212
6.8 Arithmetic Group	213
<i>Exercises</i>	224
6.9 Logical Group	224
<i>Exercises</i>	233
6.10 Control Transfer Group	234
<i>Exercises</i>	240
6.11 Miscellaneous Instruction Groups	241
6.12 Conclusion	243
<i>Exercises</i>	243
<i>Further Reading</i>	244

7. Microprocessor—Peripheral Interfacing 245–366

7.1 Introduction	245
7.2 Generation of I/O Ports	246
<i>Exercises</i>	249
7.3 Programmable Peripheral Interface (PPI)—Intel 8255	249
7.3.1 Mode 0—Basic Input/Output	250
7.3.2 Mode 1—Strobed Input/Output	250
7.3.3 Mode 2—Strobed Bidirectional Bus	251
7.3.4 Configuring Intel 8255	251
7.3.5 Interfacing the 8255	252
<i>Exercises</i>	256

7.4 Sample-and-Hold Circuit and Multiplexer	256
<i>Exercises</i>	261
7.5 Keyboard and Display Interface	261
7.5.1 Keyboard	262
7.5.2 Light Emitting Diode Display	266
7.5.3 Seven-Segment LED	267
7.5.4 The 8085 Interfacing to Keyboard and Display	271
7.5.5 The 8086 Interfacing to Keyboard and Display	272
7.6 Keyboard and Display Controller (Intel 8279)	281
7.6.1 Keyboard Section	282
7.6.2 Display Section	286
7.6.3 Software Commands	287
7.6.4 Interfacing the 8279	289
<i>Exercises</i>	313
7.7 Programmable Interval Timers Intel 8253 and Intel 8254	313
7.7.1 Operating Modes	314
7.7.2 Programming Intel 8253	316
7.7.3 Interfacing Intel 8253/8254	318
<i>Exercises</i>	335
7.8 Digital-to-Analog Converter	335
<i>Exercise</i>	345
7.9 Analog-to-Digital Converter	345
7.9.1 Asynchronous Mode	346
7.9.2 Synchronous Mode	347
7.9.3 Interrupt Mode	349
<i>Exercise</i>	357
7.10 CRT Terminal Interface	357
7.11 Printer Interface	360
7.12 Conclusion	365
<i>Exercises</i>	365
<i>Further Reading</i>	366

8. System Design Using Intel 8085 and Intel 8086 Microprocessors—Case Studies

367–417

8.1 Introduction	367
8.2 Case Study 1—A Mining Problem	367
8.2.1 Sensors	367
8.2.2 Multiplexer and Data Converter	369
8.2.3 I/O Ports	370
8.2.4 Timer Function	371
8.2.5 Alarm Circuit	372
8.2.6 Software	372
8.2.7 Using the 8085	373
8.2.8 Using the 8086	378
<i>Exercises</i>	384
8.3 Case Study 2—Turbine Monitor	385
8.3.1 Measurement Transducers	386
8.3.2 Multiplexer and Data Converter	389
8.3.3 Printer	389
8.3.4 Timer Function	389
8.3.5 I/O Ports	389

8.3.6 Software Flowchart	393
8.3.7 Using the 8085	396
8.3.8 Using the 8086	405
8.4 Conclusion	416
<i>Exercises</i>	416
<i>Further Reading</i>	417

9. Intel 8051 Microcontroller—Hardware Architecture 418–481

9.1 Introduction	418
9.2 Architecture	419
9.3 Memory Organization	420
9.3.1 Program Memory	421
9.3.2 Data Memory	421
9.4 Special Function Registers	424
9.5 Pins and Signals	430
9.6 Timing and Control	432
9.7 Port Operation	438
9.7.1 Port 0	439
9.7.2 Port 1	439
9.7.3 Port 2	440
9.7.4 Port 3	441
9.8 Memory Interfacing	442
9.9 I/O Interfacing	447
9.10 Programming the 8051 Resources	450
9.10.1 Timer/Counters	450
9.10.2 Serial Interface	456
9.10.3 Multiprocessor Communication	463
9.11 Interrupts	465
9.11.1 Response Time	466
9.11.2 Interrupt Control Registers	467
9.12 Measurement of Frequency, Period and Pulse Width of a Signal	470
9.12.1 Frequency Measurement	471
9.12.2 Period Measurement	476
9.12.3 Pulse Width Measurement	479
9.13 Power Down Operation	479
9.14 Conclusion	480
<i>Exercises</i>	480
<i>Further Reading</i>	481

10. Intel 8051 Microcontroller—Instruction Set and Programming 482–520

10.1 Introduction	482
10.2 Programmers Model of Intel 8051	482
10.2.1 Memory	483
10.2.2 Special Function Registers	484
10.2.3 Program Status Word	486
10.3 Operand Types	487
10.4 Operand Addressing	488
10.4.1 Register Addressing	488
10.4.2 Direct Addressing	488
10.4.3 Indirect Addressing	490

10.4.4 Immediate Addressing	491
10.4.5 Base Register Plus Index Register Indirect Addressing	492
10.5 Data Transfer Instructions	494
10.5.1 General-purpose Transfers	494
10.5.2 Accumulator-specific Transfers	496
10.5.3 Address-object Transfer	497
<i>Exercises</i>	499
10.6 Arithmetic Instructions	499
10.6.1 Addition	499
10.6.2 Subtraction	501
10.6.3 Multiplication	502
10.6.4 Division	502
<i>Exercises</i>	504
10.7 Logic Instructions	504
10.7.1 Single-operand Operations	504
10.7.2 Two-operand Operations	507
<i>Exercises</i>	510
10.8 Control Transfer Instructions	510
10.8.1 Unconditional Calls, Returns and Jumps	510
10.8.2 Conditional Jumps	513
10.9 Conclusion	519
<i>Exercises</i>	519
<i>Further Reading</i>	520

11. The 8051 Microcontroller-Based System Design—Case Studies 521–559

11.1 Introduction	521
11.2 Case Study 1—Traffic Light Control	521
11.2.1 Switching Circuit	525
11.2.2 The 8051 Hardware Interface	525
11.2.3 Operation Sequence	527
11.2.4 Time of the Day Clock	528
<i>Exercises</i>	538
11.3 Case Study 2—Washing Machine Control	539
11.3.1 Washing Cycle	540
11.3.2 Control System Design	541
11.3.3 Software	549
11.4 Conclusion	559
<i>Exercises</i>	559
<i>Further Reading</i>	559

12. Intel 8096 Microcontroller—Hardware Architecture 560–619

12.1 Introduction	560
12.2 Architecture	560
12.3 Memory Organization	562
12.3.1 Central Processing Unit	564
12.4 Special Function Registers (SFRs)	568
12.5 Pins and Signals	572
12.6 The 8096 Clock	574
12.7 Memory Interfacing	575
12.8 I/O Interfacing	587

12.9 Interrupts	590
12.10 Input/Output Ports	592
12.11 I/O Control Registers	593
12.12 I/O Status Registers	594
12.13 Programming of the 8096 Resources	595
12.13.1 Timers	595
12.13.2 High Speed Inputs	598
12.13.3 High Speed Outputs	600
12.13.4 Serial Input/Output	607
12.14 Multiprocessor Communications	611
12.15 Analog-to-Digital-Converter	613
12.16 Analog Output	614
12.17 Watchdog Timer	618
12.18 Conclusion	618
<i>Exercises</i>	619
<i>Further Reading</i>	619

13. Intel 8096 Microcontroller—Instruction Set and Programming 620–664

13.1 Introduction	620
13.2 Programmer’s Model of the 8096	620
13.2.1 Memory	621
13.2.2 Special Function Registers	622
13.2.3 I/O Control and Status Registers	623
13.2.4 Program Status Word	624
13.3 Operand Types	625
13.4 Operand Addressing	626
13.5 MCS-96 Assembler Directives	632
13.6 Acronyms Used	634
13.7 Data Transfer Group	635
<i>Exercises</i>	637
13.8 Arithmetic Group	637
<i>Exercises</i>	647
13.9 Logical Group	647
<i>Exercises</i>	650
13.10 Shift Group	650
<i>Exercises</i>	655
13.11 Branch Group	655
<i>Exercises</i>	658
13.12 Stack Group	659
13.13 Special Control Group	660
13.14 Other Instructions	662
13.15 Conclusion	663
<i>Exercises</i>	663
<i>Further Reading</i>	664

14. The 8096 Microcontroller-Based System Design—Case Studies 665–693

14.1 Introduction	665
14.2 Case Study 1—Numerical Control Machine	665
14.3 NC Machine Classification	666
14.4 Numerical Control Machine—Basic Design	667

14.5 The 8096-Based Control System for NC Machines	669
14.5.1 Interfacing NC Machine to the 8096	673
14.5.2 Software	675
<i>Exercises</i>	680
14.6 Case Study 2—Automation of Water Supply for a Colony	680
14.6.1 Automation of Distribution Valves	685
14.6.2 Software	686
14.7 Conclusion	692
<i>Exercises</i>	692
<i>Further Reading</i>	693
Appendix I <i>Intel 8085 Instruction Set Summary</i>	695–703
Appendix II <i>Intel 8086 Instruction Set Summary</i>	704–710
Appendix III <i>Intel 8051 Instruction Set Summary</i>	711–715
Appendix IV <i>Intel 8096 Instruction Set Summary</i>	716–721
Index	723–728

PREFACE

While the history of technology development by mankind can be considered in terms of thousands of years, the real development of technology has occurred only during the last hundred years. The last forty years represent a boom period owing to the rapid advancements of microelectronics technology and emergence of the microprocessor.

The advent of microprocessor has changed completely the system-engineering scene. It has led to a spurt in the systems design and development activity all over the world. The development of fast memory chips and other necessary hardware including semiconductor sensors, peripheral controllers, etc. has acted as a fillip to such activities in the fields where the microprocessor can be used to increase the productivity.

The microprocessors were designed for general applications and these included the applications of real-time monitoring and control as well. Soon it was realized that these sophisticated applications required a number of features not otherwise necessary in general-purpose applications. To cater to these sophisticated applications, microprocessor structures were redrawn and thus the microcontroller was born.

One of the objectives of writing this book is to impart the skill sets for system design using microprocessors and microcontrollers. Even though a number of books are available on the subject of microprocessors and microcontrollers, very few of them talk about system design concepts. During the period of my association with the Microprocessor Application Engineering Programme (MAEP) conducted by UNDP, I learnt the system design concepts in consultation with several national and international experts. Later these concepts got further crystallized during a number of training programmes on ‘System Design using Microprocessors and Microcontrollers’ organized for the industries. This book is therefore also intended to impart these skill sets to the students as well as to the industry personnel.

During the writing of this book, I did face a number of dilemmas. The first dilemma was regarding the content of the book. The 8085 microprocessor had established itself as a complete microprocessor in the late 1970s and is still considered to be an ideal microprocessor for

imparting training. Therefore, several Indian universities begin the microprocessor course with the 8085 hardware, software and interfacing exposition. At the same time, quite a few universities have changed their syllabus and commence the microprocessor course with the 8086 hardware, software and interfacing. A number of universities have also started covering the 8051 microcontroller along with either the 8085 or the 8086 microprocessor. There are also some institutions where a course in Embedded System Design covers the 8051 and 8096 microcontrollers. The present book covers the hardware architecture, the instruction set and programming, interfacing and application case studies in respect of all these microprocessors/microcontrollers.

The other dilemma, which I faced, was regarding the sequence adopted in different institutions for teaching of microprocessors. Some institutions cover the microprocessor hardware architecture first, followed by the instruction set and programming which is then followed by peripheral interfacing and applications. On the other hand, some institutions take up the instruction set and programming before teaching the hardware architecture. Considering this and to enable a wider use of the book, I have included the programmer's model in all the chapters related to instruction set and programming. Examples and exercises are part of each chapter in the book.

The book has three chapters devoted to system design case studies using the 8085, the 8086, the 8051 and the 8096. In all the case studies the descriptions of the required sensors and actuators have been covered to have a feeling of the working of the complete system as well as to have a better appreciation of interfacing and system design intricacies. These chapters have exercises as home assignments for students.

Conceptually the book may be divided into four parts.

The first part having Chapters 1 and 2 is devoted to foundation laying for subsequent chapters.

The subject of system design using the microprocessor has been introduced right in the first chapter. This chapter describes the steps to be followed by the system designer for conceptualization, design and debugging in simulated and real environments.

Chapter 2 deals with the subject of microprocessor in a generalized way. Starting with the von Neumann architecture of computers, this chapter discusses instruction execution, addressing modes, generalized instruction set, programming aspects, etc. This is followed by the generalized microprocessor architecture, microprocessor interfacing to I/O units and data transfer schemes. Architectural advancements of

microprocessor have been discussed along with pipelining, cache memory, virtual memory systems, etc.

The second part of the book covers the hardware, the software, the interfacing and the system design aspects using the 8085 and 8086 microprocessors.

Chapter 3 introduces the 8085 microprocessor hardware architecture. The memory interfacing of the 8085, the interrupt system, the machine cycles, the timing diagrams have all been described in detail. Also in Chapter 3, the state transition diagram of the 8085 processor described under Section 1.2.6 in Chapter 1 has been covered.

Chapter 4 describes the 8085 instruction set and programming. The programmer's model has been described in the introduction to the chapter, followed by addressing modes and the instruction set.

The 8086 hardware architecture has been introduced in Chapter 5. The minimum and maximum mode configurations, external memory addressing, companion chips like 8284A, 8286/8287, 8282/8283 and 8288 have been covered. Using these chips the 8086 is configured either in the minimum or in the maximum mode. At the end, a comparison between the 8086 and 8088 processors has been provided along with a compatibility description between the 8088, the 8086, the 80186 and the 80286 processors.

The next chapter, i.e. Chapter 6 describes the instruction set of the 8086 along with the addressing modes. The major assembler directives have also been described.

Chapter 7 describes peripheral interfacing with microprocessors. It covers the generation of I/O Ports, the 8255 PPI, and interfacing of the 8085 and the 8086 to sample-and-hold circuit, multiplexer, keyboard and display interface, digital-to-analog converter, analog-to-digital converter, CRT terminal, and printer. The Keyboard and Display Controller Intel 8279 and Programmable Interval Timers Intel 8253 and Intel 8254 have also been covered. It has been ensured that those students who are not taking up the 8085 can directly take up the 8086 and to that extent there is some duplication between the descriptions of an interface for the two microprocessors.

Chapter 8 covers the system design case studies using the 8085 and 8086 microprocessors. Two case studies—A Mining Problem and Turbine Monitor—have been covered in the chapter.

The third part of the book, covered in Chapters 9 to 11, relates to the 8051 hardware, software, interfacing and system design.

Chapter 9 describes the 8051 microcontroller hardware architecture. The power of the 8051 microcontroller which is evident from its hardware has been described vividly. This chapter also describes the special function registers, the memory interfacing, the I/O interfacing and the measurement of frequency, period and pulse width of a signal.

Chapter 10 covers the 8051 microcontroller instruction set and programming. The addressing modes have been described in detail along with the groups of instruction set.

Chapter 11 covers the system design case studies using the 8051 microcontroller. Two case studies—Traffic Light Control and Washing Machine Control—have been described in the chapter.

The fourth part of the book concerns the 16-bit microcontroller, Intel 8096. Its hardware architecture, interfacing, programming and system design are covered in Chapters 12 to 14.

Chapter 12 covers the 8096 microcontroller hardware architecture. The RALU architecture, memory interfacing and configuring the processor for different bus widths and bus control, special function registers, I/O interfacing, and interrupts have been described.

Chapter 13 describes the instruction set and programming of the 8096 microcontroller. The programmer's model, addressing modes, and MCS-96 assembler directives have been covered along with the instruction set.

The system design case studies using the 8096 are covered in Chapter 14. Two case studies—Numerical Control Machine and Automation of Water Supply for a colony—have been described.

Finally, the Instruction Set Summary for each of the microprocessors and microcontrollers, i.e. 8086, 8085, 8051 and 8096, has been given in a separate Appendix.

I believe that diagrams convey much more than the written text and therefore a large number of diagrams have been included to supplement the written text. I have also made special efforts to make the book modular for a variety of courses.

For the courses that cover both the 8085 and the 8086, I recommend sequential covering of the chapters from 1 to 8. While covering Chapters 7 and 8, the basic concepts may be elaborated and students may be encouraged to write software using both the 8085 and 8086. In case of courses covering the 8086 only, Chapters 1, 2, 5, 6, 7 and 8 are recommended. In this case, while covering the contents of Chapters 7 and 8, the portion dealing with 8085 may be left out. For the Embedded System Design course having the 8051 or/and the 8096, Chapters 1, 2, 9

to 11 (for the 8051) and Chapters 12 to 14 (for the 8096) may be covered.

Finally, I hope that the book would meet the students educational needs, and provide the flexibility to teachers to maximize the potential of the course.

Krishna Kant

ACKNOWLEDGEMENTS

Making this book a reality has been an arduous task. At this juncture, I deeply remember my father Late Shri R.R. Verma, my mother Late Smt. Gulab Devi and my nephew Late Shri Vivek. My parents spent their lifetime in making me what I am today. They would have been very happy on the publication of this book.

A large number of people have contributed directly or indirectly to this book. It is my pleasure to acknowledge my gratitude towards all.

The initial inspiration to transform my knowledge and experience into a book form came from Dr. N. Seshagiri, former Director General National Informatics Centre and Special Secretary, Department of Information Technology.

Close friends like Dr. Vijay Bhatkar, Chairman ETH Research Laboratory Pune, Prof. D. Popovic, former Head and Professor in the Institute of Automation Technology, University of Bremen, Germany and Late Shri G.S. Vardhan, Director, Software Technology Park of India, Bangalore, provided a welcome fillip to my writing all the time.

I received constant encouragement and friendly suggestions from Dr. Ashok Chakravarti and Dr. U.P. Phadke, both Advisors to the Department of IT, Ministry of Communications and Information Technology, which inspired my technical endeavours despite unfavourable circumstances at times.

Many students pursuing courses in Microprocessors and Applications, Microcontroller Architecture and Programming and Advanced Microprocessor Systems, organized under the New Delhi Centre of Microprocessor Application Engineering Programme (MAEP), shared their experiences with me and offered useful critique that helped me in crystallizing my concepts.

My maepian friends (even though the MAEP project has ended, the term maepian is still in use) who are currently working in various organizations continue to help me through literature survey, friendly suggestions, etc. Considerable help was provided by Smt. Uma Chauhan and Smt. Renu Gakhar.

My son Prashant Rishi (who has just completed his BE in Electronics & Communication) and my daughter Neha Shikha (who is at present

studying for her BE in Electronics & Communication) helped me immensely by going through the manuscript thoroughly.

Dr. R. Manimekalai, CPCRI Kasaragod, took active interest in my work and continuously encouraged me during my later phase of proofreading and verification.

Special mention is made of my secretarial staff who painstakingly helped me in producing the manuscript.

Krishna Kant

1

SYSTEM DESIGN USING MICROPROCESSOR

1.1 INTRODUCTION

The late 1960s saw the rapid development of semiconductor technology. Soon it became possible to manufacture complete logic circuits on a single chip of silicon. Such circuits on chips came to be known as integrated circuits, and their complexity doubled every year. Since many of the early integrated circuits were utilized in computer systems, it was natural that eventually a complete processing unit should be produced on a single chip of silicon. The world's first processing unit on a single chip, known as the Intel 4004 microprocessor, was produced in 1971. This processor was simple and it processed only 4-bit numbers, but it was the beginning of what we know today as the microprocessor. The merging of the technologies of computing and electronics in the microprocessor resulted in the establishment of a useful and powerful device that was capable of a wide range of applications, provided that suitable software could be developed.

Microprocessors soon reached performance levels that took mainframe computers 30 years to achieve. In terms of net computing power delivered by the industry, the microprocessor of today outperforms the mainframes and the minis of yesterday. Microprocessors have thus influenced our society more pervasively than mainframe computers, despite the two decades of lead that the latter enjoyed. The microprocessor computing power became more and more affordable, affecting the ways we communicate, access information, manufacture our products or conduct our business.

Today, microprocessors are critical components in a wide range of systems and equipment including consumer products such as automobiles, washing machines, televisions and video recorders as well as industrial and commercial equipment. Their range of applications

continues to grow as societies are becoming increasingly dependent upon information storage, transfer and retrieval.

1.2 SYSTEM DESIGN

The design of microprocessor-based systems involves several activities ranging from feasibility study to field trials and documentation. The system development cycle is depicted in Figure 1.1.

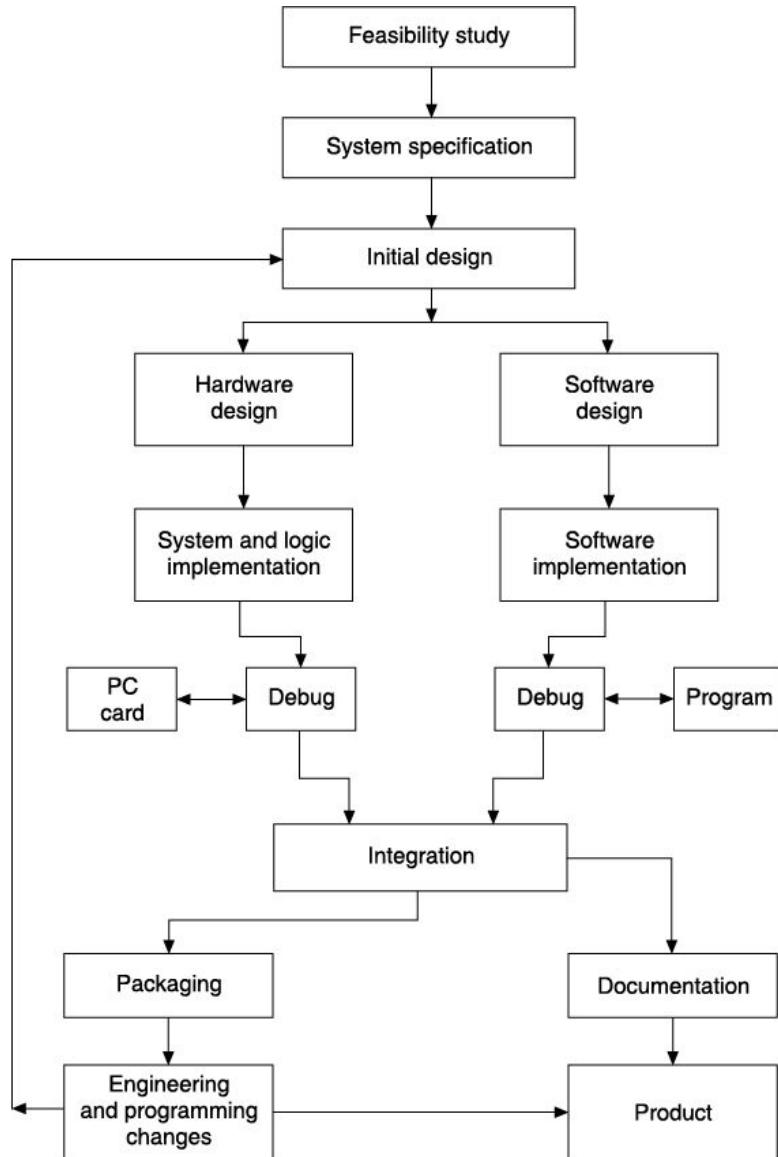


Figure 1.1 System development cycle.

1.2.1 Feasibility Study

The feasibility study provides the background for actual system development. It helps in the selection of a particular level of technology depending on the availability of expertise to operate and maintain the

end product. It also analyses various approaches like purchasing a system and modifying for a particular requirement or designing the system from PCB or component level. It also provides cost-time analysis.

However, the fundamental question we must ask ourselves after the application has been understood and before the start of system design, is: do we need a microprocessor? This is the most crucial decision the system designer has to take. We should not forget that despite the power of the microprocessor, in many cases a hardware solution might be better and cheaper. We should avoid selecting the microprocessor just for fancy.

Figure 1.2 explains the alternatives available to the system designer to develop his system. Clearly, if the system is not complex and the number of units being produced is small, the random logic will be most suitable. The Application Specific Integrated Circuit (ASIC) or the custom LSI will be suitable for a large number of units but with lesser complexity.

The microprocessor-based design is a *compromise* between random logic and custom LSI alternatives. The microprocessor-based design may appear to be more expensive than the random logic alternatives because of the high initial cost of software development involved. However, the initial cost is offset by cheaper components and simple hardware circuitry, which make the microprocessor solution eventually less costly over a certain range of production volumes.

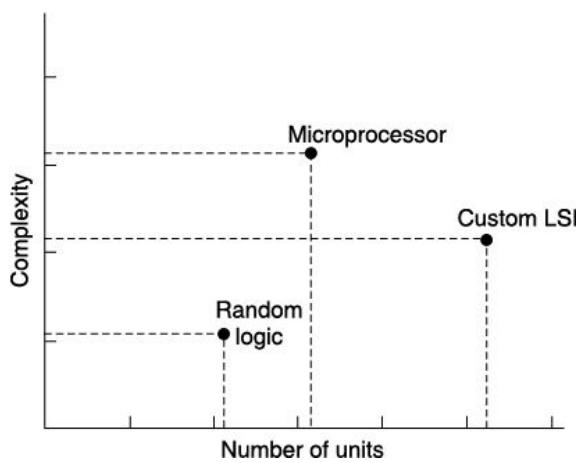


Figure 1.2 Different alternatives available to the system designer.

1.2.2 Random Logic vs. Microprocessor

Random logic offers the design advantage over the microprocessor-based system when one or more of the following are applicable.

- The functions to be performed are minimal.
- The input and the output consist of a single channel.
- The system operates on only one function at any time (though there may be multiple inputs) or the system has a single word transmission structure.
- A small system has to be custom-designed.
- High-speed operation is required.

Microprocessors offer advantages over random logic when one or more of the following are applicable.

- Software can be traded off for additional hardware, so that the system capabilities can be expanded readily without system redesign.
- Multiple inputs are needed.
- A large number of functions must be performed.
- Multidecision paths are required.
- Large memories are involved.

The following “Rule of Thumb” has been often used to determine the cost trade-off between random logic and microprocessor.

A 50 IC random logic system costs about the same as a microprocessor plus 20 interface ICs. Included in this rule are the costs of components, handling, testing, and interconnections.

A flowchart depicting the selection between random logic and microprocessor is shown in Figure 1.3.

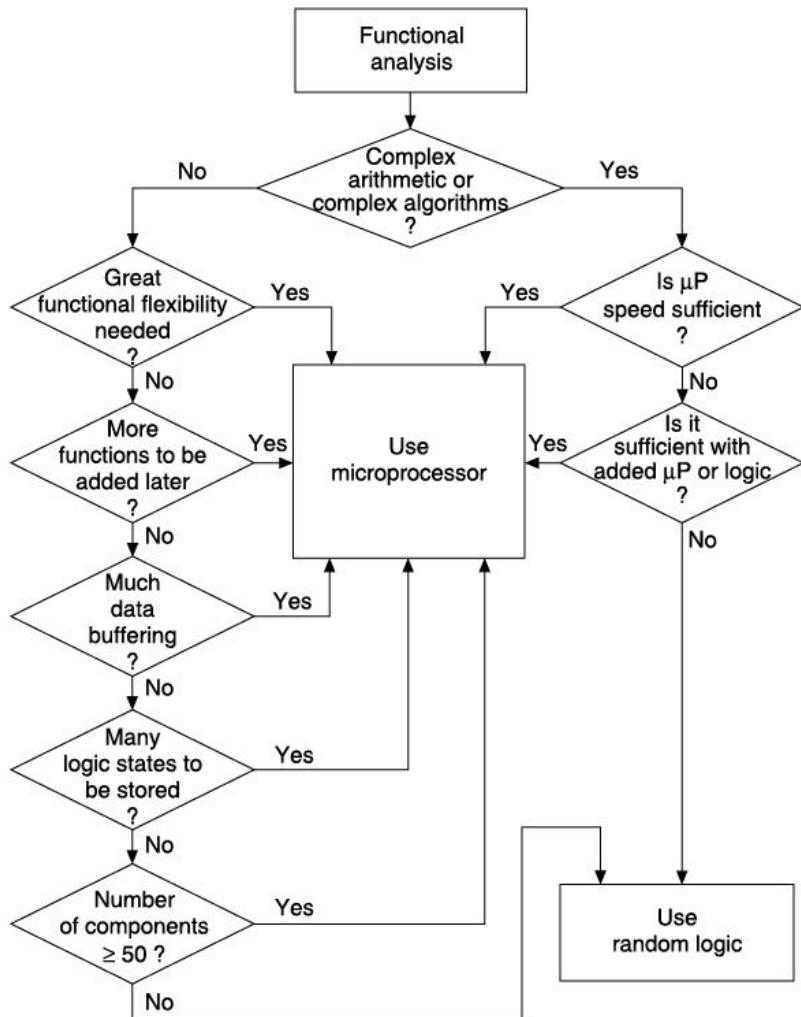


Figure 1.3 Flowchart depicting the selection between random logic and microprocessor.

If after considering all the points, the microprocessor-based approach is taken, a suitable microprocessor has to be selected. The technical characteristics such as speed, word length, addressing modes, number of scratch pad registers, single bit manipulation, instructions set, arithmetic capabilities, etc. should be considered with respect to the system requirement for the selection of a suitable microprocessor. The designer has also to decide between microprocessor, microcomputer and microcontroller based on the application requirements. Microcontrollers are preferred for all embedded applications since they have bit manipulation instructions, built-in memory, I/O ports as well as powerful instruction sets.

1.2.3 System Specification

This is the stage of system development in which we define our problem and the ultimate system that will solve our problem.

A system can be defined in terms of a set of entities and the relation between entities. The system under consideration may itself be working as a subsystem of some larger system. In that case, the system will be accepting inputs and providing the outputs to the larger system. These inputs and outputs are called the environment of the system. A crude way of defining the system is by defining the inputs and outputs from the system, but a systematic designer will break the system into different functionally independent subsystems and define the environment of each of these subsystems (Figure 1.4). This methodology of paying more attention during the system definition phase is always helpful during the later period when some changes are required.

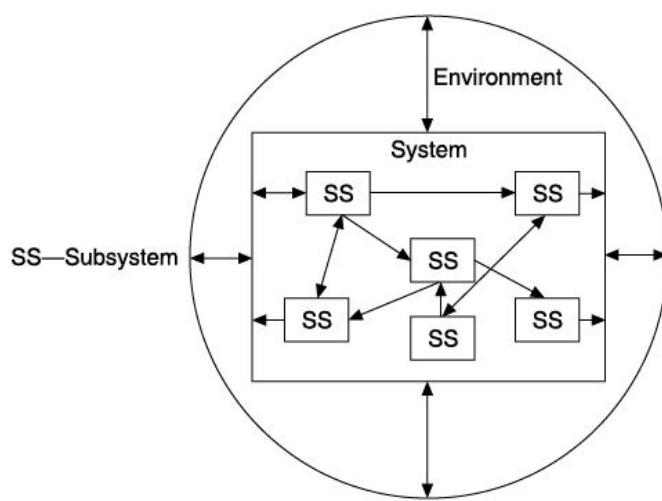


Figure 1.4 System and environment.

It is worthwhile quoting here the Murphy's law of random system design, which states that the following things are likely to be true if complex problems are tackled in a random way.

- Things are more complex than they seem to be.
- Things take longer than the expected time.
- Things cost more than the expected cost.
- If something can go wrong, it will.
- The output of such an exercise may be FRUSTRATION.

The Callahan's corollary to Murphy's Law is "Murphy was an optimist". Thus the sound definition of the system will lead to an optimum design. Moreover, since more than one system engineers are usually involved in the development, the interface between these engineers will be facilitated by adopting the organized approach.

It is here that the documentation should be initiated. The initial document will further get refined and the details will be added at every stage of development.

1.2.4 Initial Design

The initial design defines the functions that will be carried out by both hardware and software. An analysis of the problem should be done to define performance requirements, basic hardware configurations, basic software routines, etc. If a simulation facility is available, it should be made use of, to make the initial design as perfect as possible. The initial design will help in estimating the memory requirement, and other timing considerations.

The selection of a particular hardware configuration will strongly influence the software design. Though the adoption of software in place of hardware will definitely reduce the system cost, a clear consistent structure of the hardware should not be compromised. The implementation of simple hardware functions can often help considerably in reducing the software. There are certain thresholds beyond which software development becomes very complex and expensive. For example, if the software increases slightly over 1 kilobyte (abbreviated KB) or if more I/O lines than those provided in an I/O port are required, a full additional memory chip or I/O port has to be added, thus increasing the hardware cost.

1.2.5 Hardware Design

The matching of electrical characteristics and timings of ICs may create problems during hardware design. The simplest solution (though not the cheapest) will be to use a complete family of microprocessor and peripheral chips for which these conditions are satisfied. When using a different manufacturer or series, some hardware adaptation may become necessary. Similarly, if the components of different logic families are interfaced, additional hardware (i.e. driving circuits, pull-up registers, etc.) might become necessary.

1.2.6 Software Design

The software design basically involves the following steps.

Environment and problem specification

Though the overall system specification is given at the beginning of the project design, the parameters relevant to the software design should be

documented separately for the software group. This specification should contain a brief statement of the problem, operating characteristics needed for programming, for example, data format, data transfer modes between peripherals, etc. and the software environment of the system.

State transition diagram

The state transition diagram is a finite state model of the system, which shows all possible states that the system can achieve. This is very useful in interactive systems with external excitation such as a keyboard-controlled monitor etc. It also shows the external inputs that can be received by the system to switch from one state to the other.

We will now evolve some of the symbols and definitions which will help us in drawing and understanding the state transition diagrams.

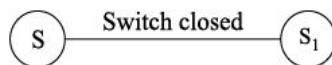
Primary excitation: External inputs like switch closed etc.

Secondary excitation: Internal inputs as a result of the reactions within the system due to external inputs.

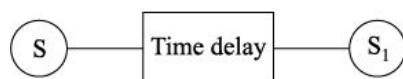
State: The system stays in state S until new excitation occurs.



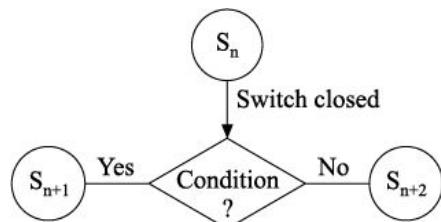
Transition: (i) From state S to S_1 due to primary excitation ‘Switch closed’.



(ii) From state S to S_1 due to secondary excitation ‘Time delay’.

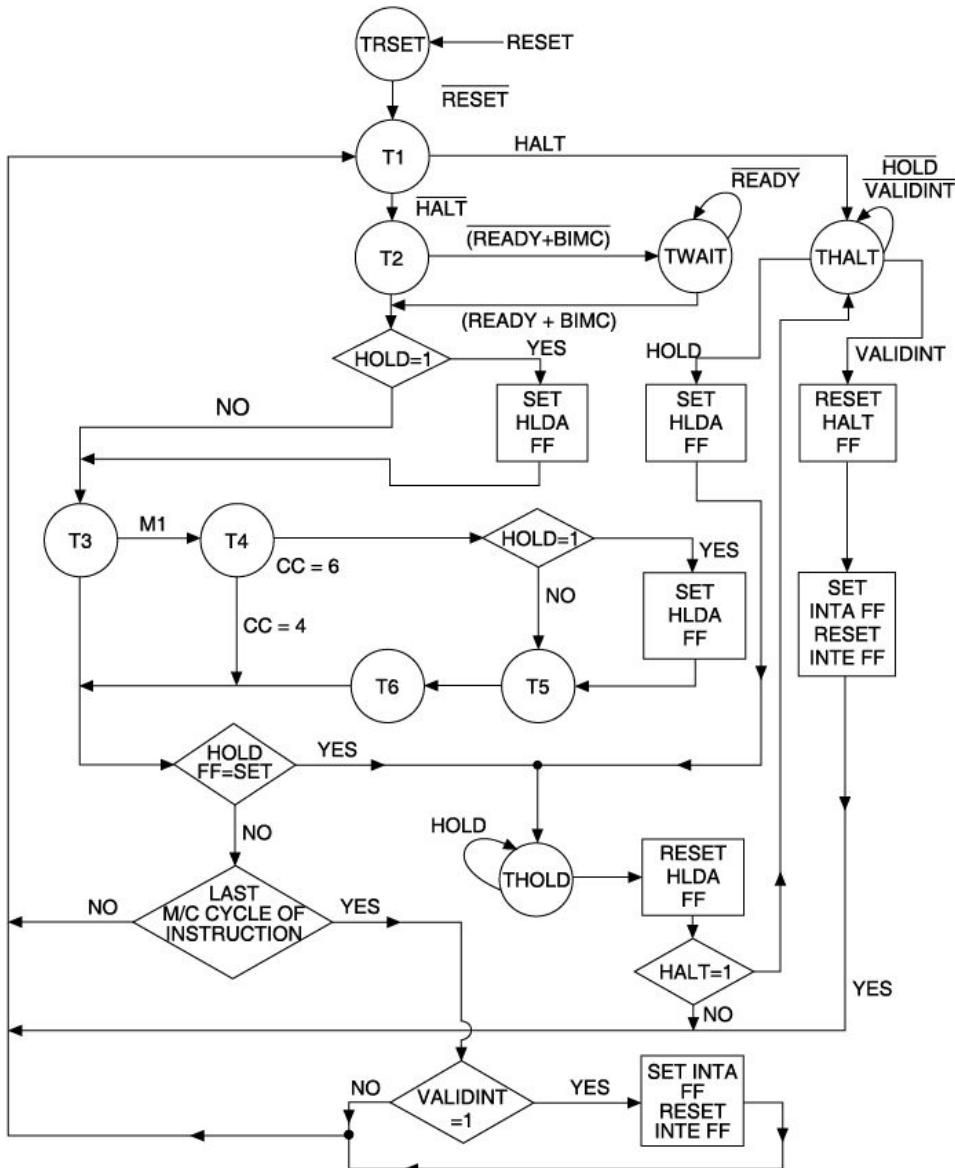


Decision symbol: Depending upon the internal condition, the transition can occur to one out of several states upon primary excitation ‘Switch closed’.



The state transition diagram of the Intel 8085 CPU showing the sequence of all possible states which the CPU can assume when

executing an instruction is shown in Figure 1.5. This shows the use of a state transition diagram for hardware/firmware implementation.



Symbol Definitions

CPU state T. All CPU state transitions occur on the falling edge of the CLK.

A Decision (') that determines the switch of several alternative paths to follow.

Perform the Action X.

Flowline that indicates the sequence of events.

Figure 1.5 State transition diagram of the 8085 CPU (contd.).

Flowline that indicates the sequence of events if condition X is true.

- CC Number of clock-cycles in the current machine cycle.
- BIMC “Bus Idle Machine Cycle” = Machine cycle which does not use the System bus.
- VALIDINT “Valid Interrupt” = An interrupt is pending that is both enabled and unmasked (masking only applies for RST 5.5, 6.5 and 7.5 inputs).
- HLDAFF Internal hold acknowledge flip-flop. Note that the 8085A system buses are tri-stated one clock-cycle after the HLDA flip-flop is set.

8085 Machine State Chart

Machine	S ₁ , S ₀	Status and Buses				Control		
		IO/	A ₈ -A ₁₅	AD ₀ -AD ₇	*, #	-	-	ALE
T1	□	□	□	□	1	1	1	1
T2	□	□	□	□	□	□	□	0
TWAIT	□	□	□	□	□	□	□	0
T3	□	□	□	□	□	□	□	0
T4	1	0*	□	TS	1	1	1	0
T5	1	0*	□	TS	1	1	1	0
T6	1	0*	□	TS	1	1	1	0
TRESET	□	TS	TS	TS	TS	1	1	0
THALT	0	TS	TS	TS	TS	1	1	0
THOLD	□	TS	TS	TS	TS	1	1	0

0 = logic “0”, 1 = logic “1”, TS = high impedance, □ = unspecified, ALE not generated during 2nd and 3rd machine cycles of DAD instruction.

*IO/M = 1 during T4-T6 states of RST and INA cycles.

Figure 1.5 State transition diagram of the 8085 CPU.

Program schedule

In a real time application, the time constraints and the program sequence play an important role, therefore, in such cases the program schedule can be very useful. It should show the programming sequence, timing consideration of the system and the places where the time criticality may occur.

Hierarchy model

A hierarchy model represents (Figure 1.6) the various levels of modules in the program. Though it gets modified from time to time during the total design cycle, it is very necessary to work out an initial hierarchy for the software module. It is useful for the design of complex systems like

monitor, file management, etc. in which case a module will pass control (by subroutine branching) to other modules to execute various functions.

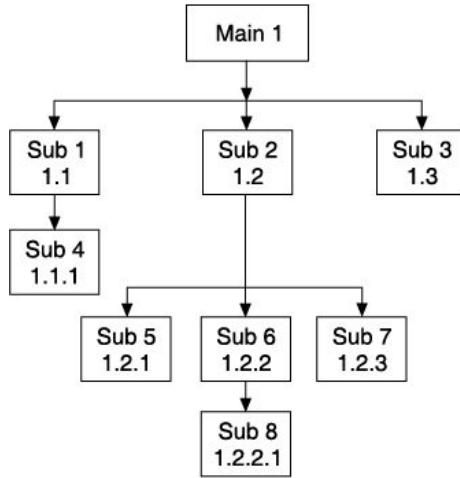


Figure 1.6 Hierarchy model of software.

Flowchart

In software design, one should preferably follow the top-down approach in which the designer has to define all the program requirements before attempting the implementation. In a bottom-up approach, on the contrary, one starts at a very low level of detail, thereby often losing sight of the overall objectives.

A flowchart is a graphical representation of the problem involving activities, operations and decisions in a logical and sequential manner. Thus conforming with the top-down approach of system design, a program structure composed of main program blocks should be designed. Each block, then, should further be refined. Thus several flowcharts can be drawn in increasing level of refinement.

Memory, I/O and register organization

A general memory map should be made which will show the various memory blocks, their capabilities, starting and ending addresses. Then, on that basis, separate maps for RAM and ROM area allocation should be prepared. These maps should show the detailed organization with the definitions of bytes and bits.

Similarly, the I/O organization should be planned for the mapping between the hardware I/O pins and the software bits. The I/O organization should contain the I/O addresses, the port numbers, the data formats, etc.

In the same way, the various variables assigned to the registers must be defined. This is to find out the optimum utilization of registers. The

designer can also make use of a stack to store the values of registers, temporarily.

Editing and assembling

The procedure of writing a program is called editing. It is done manually if the microprocessor kit or the microcomputer does not have an editor. With the microprocessor development system the editing is facilitated by the resident editor software. The program is then assembled, i.e. it is translated into object codes. This job is either done by the assembler if it is available with the machine or otherwise performed manually.

A number of cross assemblers are available for almost all microprocessors on the Internet. These can be used for editing and assembling the program on personal computers.

1.2.7 Test and Debug

As opposed to the software development, the bottom-up approach should be used for testing, i.e. first the smallest units, and then the larger blocks, and finally the complete system is tested. The testing should be done preferably by simulating the environment in which the module is to work. Because of the bottom-up approach, the errors at the lowest-level module can be detected and rectified. Thus the lower modules cannot act as error sources to higher-level modules.

The microprocessor simulator software available on the Internet can be used to test the assembled software on personal computers. However, a microprocessor development system is best suited for this purpose.

The errors in a microprocessor-based system can occur not only due to hardware and software, but also due to the interaction between the two. The system test starts with hardware. It is assumed that the peripheral hardware test has already been carried out independently. The system hardware can best be tested by using small and simple programs for testing each individual function. Once these tests are successfully completed, the actual software is tested block by block. When testing individual software blocks, it should be made sure that the transfer parameters satisfy the conditions of the integrated software.

1.2.8 Integration

Once the testing of hardware and software subsystems is completed, the integration of hardware and software is carried out and then the system is tested under real/simulated conditions. It needs to be emphasized here that due to various problems the integration of the two subsystems may

not be smooth, particularly if the software and the hardware have not been developed in a coordinated and organized manner.

1.2.9 Documentation

Documentation of the system is as important as system development. However, it is often overlooked and even de-emphasized by designers. For this empathy a heavy price is often paid when a fault occurs or when one of the designers leaves the organization. All the activities of both hardware and software should be documented during the development period itself. Documentation helps in the coordinated approach to system development. The system specifications, the environment, the hierarchical chart, the flowchart, the state transition diagrams, the test procedures, etc. should all be included in the documentation, to facilitate system development and its subsequent use.

1.3 DEVELOPMENT TOOLS

A number of approaches can be used to simplify the development of microprocessor-based systems. Each of these depends upon the availability of suitable development tools. The four main approaches used are:

- (a) Microcomputer kit
- (b) Dedicated microprocessor development system
- (c) Universal microprocessor development system
- (d) Software package/simulator.

1.3.1 Microcomputer Kit

The main purpose of a microcomputer kit is to impart training on the microprocessors. The use of a microcomputer kit to develop application software is perhaps the most crude and error-prone of all the methods. A typical microcomputer kit essentially consists of the central processing unit of the target machine on which the application program in binary code is executed, a small amount of the random access memory, serial input/output ports for interfacing to the peripherals, interrupt facility, and a quartz crystal controlled clock, etc. The system program consists of a simple monitor, an assembler in some cases and a debugger. Such a kit can only be used for the development of simple programs written in machine code or assembly language. The debugging facilities provided are very limited. So it is very difficult to develop an application program by using the microcomputer kit.

1.3.2 Dedicated Microprocessor Development System

A dedicated microprocessor development system (MDS) is a microcomputer system specially designed to aid in the development of a prototype system incorporating a specific microprocessor. It incorporates various user aids and test programs that enable a prototype system to be fully analyzed. As implied by the term dedicated, such systems can only be used to develop software for one or a limited range of microprocessors manufactured by the same manufacturer. The main drawback of an MDS system is that the user will have to base all his future systems on one particular microprocessor.

A typical microprocessor development system will have the following configuration.

- A central processing unit
- Random access memory upwards from 16 KB
- A visual display monitor
- An alphanumeric keyboard
- A printer
- A secondary mass storage device, i.e. either a floppy disk or a hard disk
- Ports for interfacing other peripherals
- Hardware debugging aids
- PROM/ EPROM Programmer, UV eraser
- An in-circuit emulator
- Slots for additional boards.

The heart of the system which actually facilitates system integration and debugging is the in-circuit emulator (ICE). The hardware debugging aids include Logic State Analyzer, Signature Analyzer, etc. We shall describe these facilities in brief in the following sections.

Logic state analyzer

When errors occur during data transfers or interrupt servicing, it is often necessary to examine a number of the streams of data which have been handled. If we think of an 8-bit microprocessor, there are 16 address lines and 8 data lines in addition to the control lines to investigate. So a single- or twin-beam oscilloscope is not a suitable tool for this purpose. An instrument developed particularly for this situation is the Logic State Analyzer. It has 16–48 data probes which can be attached to the bus lines

to capture data on the rising edge of each clock pulse. Generally, a block of 250 successive bits are captured by each probe and stored in a semiconductor memory. The sequence is started by a trigger pulse which can be either an electrical signal such as an interrupt or a timing pulse or a derived form of a particular bit pattern which appears on the line under test. Once captured, the data can be displayed either as a sequence of words coded in hexadecimal, octal or binary format or as a set of waveforms.

Where the program appears to jump out of the intended sequence, the probes can be put on the address lines with the sampling enabled by the Valid Memory Address (VMA) signal. The data captured will then be a list of addresses from which the microprocessor has fetched successive instructions and data. If any interface package appears to be working incorrectly, its address can be used as the trigger signal and the data lines can be examined for the correct signals.

Although this procedure gives a useful record of the data flow and addresses used, it is essentially a sampled system. Any short-duration transients which occur between the clock pulses could cause false data to be received, though not perceivable by the display. In order to check for this, it is usually possible to operate the analyzer asynchronously. In this mode, the data sampling block is derived in the analyzer and is not locked to the microprocessor clock. By running the sampling clock 5–10 times faster than the microprocessor clock, transients which are long enough to cause a logical error will normally be sampled and stored and also seen on the display.

Signature analyzer

The signature analyser is an important tool for hardware debugging. This was originally developed as a quick method of checking the complex digital instruments but is equally applicable to the microprocessors as well. This allows a stream of data to be checked easily for any single-bit error. The process is similar to the one that is used to generate the check character, normally sent after each block in many kinds of digital transmission, and uses a shift register with feedback connections.

By itself the signature conveys no meaning; it is used in a comparison process by first measuring the signature of a suitable set of nodes for a set of standard input signals in a system known to be working correctly. These form a standard signal for the system. By measuring the signature at successive nodes of a faulty system, it is possible to locate the fault within some particular section of the equipment. It should be noted that

however long the train of data signal may be, if only one bit is in error, the signature obtained will differ markedly from the correct one. The test thus has a high discrimination capability.

PROM/EPROM programmer and UV eraser

Once the application software has been developed and checked in the prototype hardware module, the next task is to store the software. It can be either in a disk, floppy, or in semiconductor memory. In an application where the software does not require frequent alteration or modification, it is better to load the program in the semiconductor memory (PROM/EPROM). PROM is a programmable ROM and once it is programmed, i.e. data is stored in it, the same cannot be erased. But in an EPROM, though the data is stored in the same way, it is erasable and we can store the new data after erasing. The ultraviolet rays are allowed to fall on the small window of the EPROM chip to erase the previously stored data. The UV erasers and EPROM programmers are available commercially.

In-circuit emulator (ICE)

The ICE is used to combine software testing with hardware testing. The MDS with the ICE option contains at least two microprocessors. The MDS processor supervises the system resources, executes the system monitor commands, and drives the system peripherals. The second processor, i.e. the ICE processor, interfaces directly to the designer's prototype or production system via an external cable.

In-circuit emulation is carried out by pulling out the socket-mounted microprocessor chip from the prototype microcomputer system for which the software is being developed and replacing it by the in-circuit emulator. Thus the hardware system to be tested has access to all the resources of the MDS and can be tested in real time. As the system operates in real-time, the response of the input/output system to particular events can be tested by interrupting program execution and by checking the contents of memory locations and address registers.

Sometimes it may be required to emulate the system more precisely. In real-time emulation we may not get the detailed trace data. The single step facility is usually available to get the detailed trace data.

This single stepping differs from what is available in a microprocessor kit. Here, the number of steps to be emulated can be programmed. We can also specify the particular conditions for halting. At the end of emulation, the debugged code can be stored as well. This facility of combined hardware and software testing of the target system is

extremely useful and can considerably shorten the system development time.

The MDS also includes considerable software facility. Typical software modules included in MDS constitute a single or multitask disk operating system which contains

- Assemblers
- High-level language compilers/interpreters
- Loaders
- Linkers
- Text editors
- Debug monitors
- File handling/management routines.

An operating system provides interface between the hardware and the user. It consists of the set of program modules, as mentioned above, for allocating and controlling the resources available on the system automatically.

1.3.3 Universal Microprocessor Development System

A dedicated MDS can only be used for developing the microcomputer hardware/software for a single microprocessor or a few of the same series manufactured by the same company. Though it is true that majority of the applications can be implemented using any of the available microprocessors, efficiency wise, some microprocessors may be superior to the others. So the MDS manufacturers have come forward with a universal MDS, which can be used to develop and test the hardware/software for a number of different systems. The term “universal” is rather misleading because no single development system can be used to develop and test the hardware/software for all the available microprocessors. However, manufacturers do provide an attractive method of developing the hardware/software for a wide range of microprocessors provided by a number of different suppliers. In addition to all the normal facilities available in a dedicated system, the universal system has the facility to plug in different modules which result in the development system emulating the behaviour of a particular microprocessor.

1.3.4 Simulators

Simulator is an alternative to the MDS for the development and testing of the microcomputer software. As the term implies, simulators are used to simulate the features, such as memory and registers of a target microprocessor on a host computer (personal computer). The application program object code for the target microprocessor, generated by a cross compiler or a cross assembler is executed on the host computer system. As the execution proceeds, the contents of the simulated registers and memory locations are altered as they would be in the real system. These values can then be printed out and analyzed. Here the execution takes place in simulated time and not in real time. The distinct advantages of simulators are given below.

- (a) As there are no real-time execution constraints, a more thorough check can be carried out on the program output thereby helping with the process of debugging the program.
- (b) It is also possible to evaluate the likely performance of a microprocessor for a particular application.

The main drawback lies in the difficulty in simulating the effect of input and output values. Testing and debugging should be carried out in real time, being not possible on a simulator.

1.4 CONCLUSION

We have described the steps that a microprocessor-based system designer should follow for conceptualizing, designing and then testing the system in simulated as well as in real environment. The design will involve selecting a microprocessor and then developing a system around that by interfacing the microprocessor to memory and peripheral devices required for the applications. The software development to execute the tasks of the applications using the microprocessor must be carried out as a parallel exercise preferably by a different team. This should be followed by integration and testing activities related to the system. Documentation at every stage should be considered as a mandatory requirement.

In the subsequent chapters when we study the microprocessor hardware, interfacing, software development and case studies, all these concepts will further get exemplified.

EXERCISES

1. Suppose we have to design and implement a system for display of arrival/departure of flights in an airport. Draw the complete system design cycle along with the information to be collected, the alternatives available and the decisions made at each step.
2. For dashboard control in a car, an electronic system needs to be designed. What will be your design strategy—to use random logic or microprocessor? Give reasons.
3. Why must the software development activity be top-down and software debugging be bottom-up? Explain with the help of an example.
4. What is a microprocessor development system? Elaborate its features.
5. Elaborate the differences between a simulator and an emulator. What is the function of the in-circuit emulator? Why the same function cannot be achieved through a cross assembler-cum-simulator?

FURTHER READING

- Bhatkar, Vijay P. and Krishna Kant, *Microprocessor Applications for Productivity Improvement*, Tata McGraw-Hill, 1988.
- Krishna Kant, *Microprocessor Based Data Acquisition System Design*, Tata McGraw-Hill, 1987.
- Rafiquzzaman, Mohammad, *Microprocessor and Microcomputer Development Systems*, Harper & Row, New York, 1984.

2

WHAT A MICROPROCESSOR IS

2.1 INTRODUCTION

Today the microprocessor-based products have revolutionized every area of electronics and have made a deep impact on quality of human life. Right from cell phones to washing machine to most sophisticated supercomputers, microprocessor technology has been the driving force for developments that have taken place over the last 35 years. Even though the microprocessor evolved from the developments in computer, it is now a totally separate area of technology.

In this chapter we take a look at the evolution of microprocessor technology and also the functions which we expect to be fulfilled in any microprocessor. The computer technology has been described in brief as it is the forerunner of microprocessors. This is followed by a description of advancements in microprocessor technology. Such a historical background will help in understanding many concepts that would be encountered in later chapters.

2.2 COMPUTER AND ITS ORGANIZATION

Figure 2.1 illustrates the block diagram of computer showing different units. It is known as von Neumann organization of computers. The computer organization proposed by von Neumann envisages that the binary number system to be used for both data and instructions. We shall now describe the different units used in the computer organization.

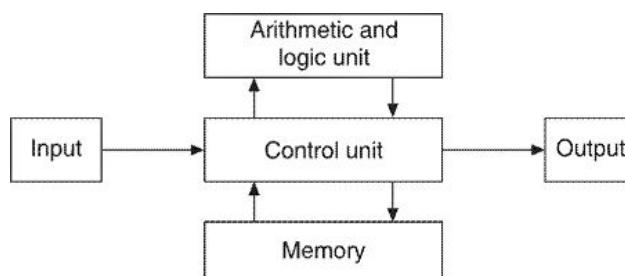


Figure 2.1 Computer organization.

2.2.1 Input System

The role of the input system is to enter instructions (for the operation to be performed) and the related data into the computer. The keyboard is the most widely used input device for entering data and instructions directly into the computer.

Other non-conventional input devices used in control applications are toggle switches, transducers, limit switches, etc.

2.2.2 Output System

The output system presents to the user the information inside the computer. The information may be input data, final result or intermediate calculations, histogram plot, etc. in data processing applications. It could be valve positions, alarm conditions, limits, event marking etc. in a real-time control environment. The output devices range from line printer, video display unit, alarm annunciator, LED/LCD display, to mimic display etc.

2.2.3 Arithmetic and Logic Unit (ALU)

The ALU performs arithmetic and logic functions on the data. The basic arithmetic functions are add, subtract, multiply, and divide. The logic functions are AND, OR, NAND, NOR, NOT, EX-OR, etc.

The main constituents of ALU are shown in Figure 2.2. The data registers provide temporary storage to data during computation. A register is a hardware device which stores data values of N bits. A computer which can store N -bit data and compute on N -bit data will have N -bit registers. Accumulator is a special data register through which calculations are performed in most of the computers. It stores one of the operands while the other operand may be stored in another data register or memory.

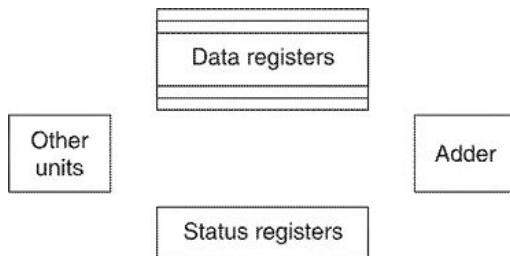


Figure 2.2 Arithmetic logic unit.

The adder is an essential part of ALU which is used to perform addition operations on two numbers. The other arithmetic operations can be performed through addition. For example, subtraction can be performed by 2's complement addition. Multiplication can be looked as repetitive addition and division as repetitive subtraction.

However, the present day computers have a full-fledged multiplier and divider unit. The arithmetic units also have capabilities to perform operations on floating point numbers. Many computers have several arithmetic units for fast computations on arrays of numbers.

2.2.4 Memory

Memory stores the bulk of data, instructions and results. The basic unit of memory is called *word*. A word can store one unit of data. The maximum number of bits of data that a word can store is called its length. Eight-bit words are known as bytes. Computer memories are of two classes—main memory and secondary or back-up memory.

The main memory is a random access memory in which any particular word can be read or written at random. The time taken to read/write different data from/to different locations in this class of memory remains the same.

Each memory location is associated with an address, which is used to access that particular location. With every read or write operation on the memory, the address where the operation is to be performed is also specified. For example, as shown in Figure 2.3, if we were to write 03, 04, and 05 at the locations with addresses 570, 571 and 572 respectively, both data and address would need to be specified with each write instruction. The memory read/write operations are performed using two registers, namely Memory Address Register and Memory Buffer Register (MBR). The MAR contains the address where the read/write operation is to be performed, whereas MBR contains data to be written or data after the read operation.

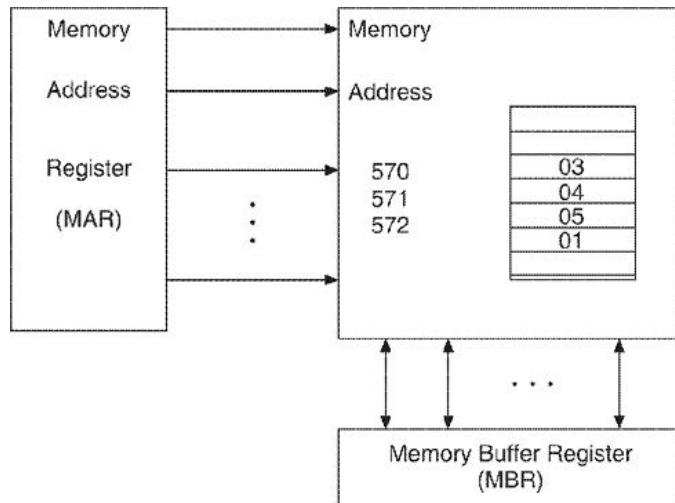


Figure 2.3 Main memory read/write operation.

Evolution of main memory has come a long way from mercury delay line, magnetic core to metal oxide semiconductor (MOS) memories. Though MOS memories are volatile (i.e. the contents are lost when power is switched off) as against nonvolatile magnetic core memories, they are extremely compact and take much less time for read/write (in nanoseconds as against microseconds in core memories). The two principal types of MOS memory chips are EPROM—Erasable Programmable Read Only Memory—2708 (1K-byte), 2716 (2K-byte), 2732 (4K-byte), 2764 (8K-byte) etc. and RAM—Read/Write Memory—6108 (1K-byte), 6116 (2K-byte), 6132 (4K-byte), 6164 (8K-byte) etc.

Secondary memories are used for large data storage. They are nonvolatile, semi-random or sequential access and have low cost per bit. Magnetic hard disks, magnetic floppy disks, magnetic tapes, magnetic drum, charge couple devices, magnetic bubbles and compact disks are amongst some of the common secondary memory devices. Each of these devices uses a different technology. The floppy disk and the compact disk are the most popular secondary storage devices in use today in personal computers.

2.2.5 Control System

The control unit is the central nervous system of the computer. It reads an instructions from memory and generates control signals for the execution of that instruction (Figure 2.4). The instruction can be to perform a read/write operation on memory or on an I/O device or to perform an arithmetic/logic operation on data.

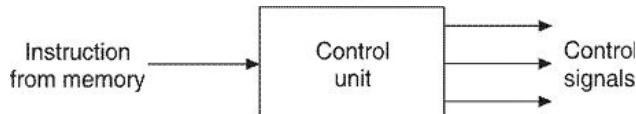


Figure 2.4 Control unit.

The data may be stored in some register of ALU (previously read from memory or result of some other operations) or it may be in memory itself. The control signals initiate the required actions in ALU, memory and I/O devices.

The program counter contains the address of the current instruction in memory and is incremented during the instruction execution. The Instruction Register stores the instruction read from memory.

Since the instruction received by the control unit is a bit pattern, it should be decoded to generate a particular sequence of control signals required. The instruction decoder is a basic and necessary part of the control unit (Figure 2.5).

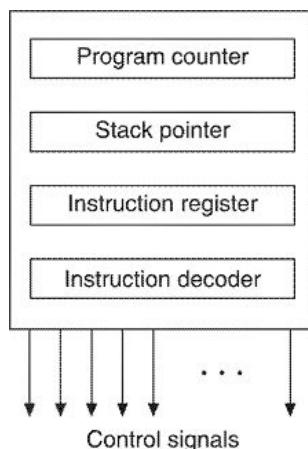


Figure 2.5 Control unit organization.

Stack

The control unit maintains a stack in CPU or main memory. Stack is like a tunnel whose other end is closed, with the result that the last entry will make the first exit. Because of this, stack is also called Last-In First-Out (LIFO) queue. The Stack Pointer (SP) points to the next available memory location in the stack (Figure 2.6). There are two operations which can be performed on stack. They are PUSH (write data into stack) and POP (read data from stack). Since stack is LIFO, the operations on stack are performed from top. Figure 2.6 shows typical PUSH and POP operations. The IN and OUT in the figure are input and output buffers for stack operation. For example, before initiating the PUSH operation, the data (Z in our example) to be pushed should be stored in IN buffer. Similarly, after the POP operation has been completed, data taken out from top of the stack will appear in OUT buffer. These IN and OUT buffers are normally some registers in the ALU. One may wonder at this juncture about the real utility of stack in the computer, but we will find it extremely useful while dealing with subroutines, interrupts and other real-time operations.

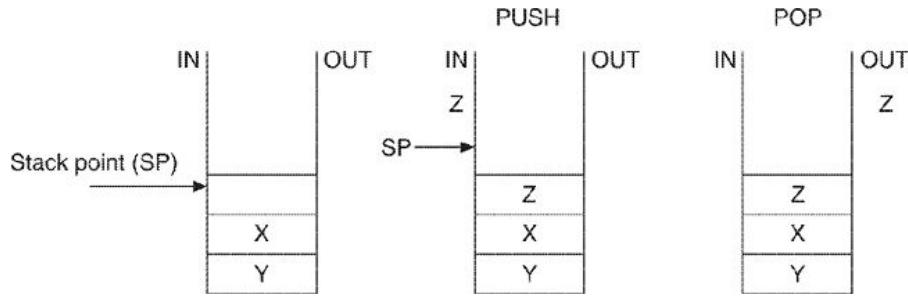


Figure 2.6 Stack.

2.2.6 Instruction Execution

Having dealt with the basic subunits in control unit, let us now discuss instruction execution. The instruction execution includes two basic steps

1. Instruction fetch, i.e. reading of instruction from memory
2. Instruction execute, i.e. generation of control signals.

These two steps are called fetch cycle and execute cycle. Thus, an instruction cycle of the control unit consists of a fetch cycle and an execute cycle. The operations in fetch and execute cycles are shown in Figure 2.7.

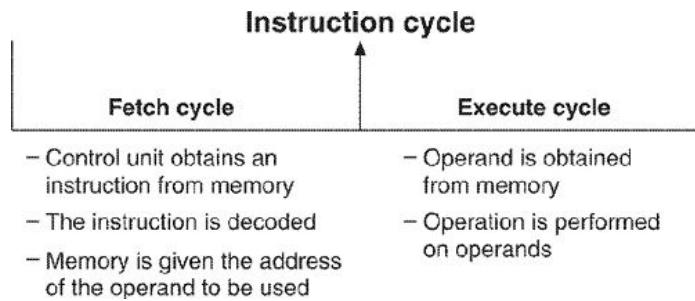


Figure 2.7 Instruction execution.

2.2.7 Clock

All operations in the computer are controlled by clock signals. The clock signal provides the basic timing signal for computer operations. In computers, the clock period may vary from few nanoseconds to few microseconds. As shown in Figure 2.8, there are two edges (leading edge and trailing edge) and two states (level 0 state and level 1 state) per period. All the actions in the computer are initiated either by the leading edge or by trailing edge of the clock and take a fixed number of clock periods to complete.

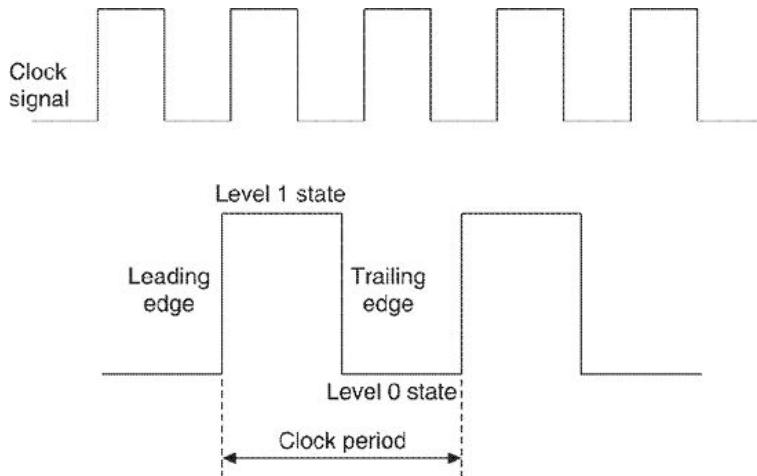


Figure 2.8 Clock pulse.

As an example, the instruction fetch cycle will take a fixed number of clock cycles. As part of instruction execution, one or more of the following operations may be involved.

- Read from memory.
- Read from input device.
- Write to memory.
- Write to output device.

These operations will also take a fixed number of clock cycles. The internal execution of an instruction, e.g. transfer of data from one register to another, addition, subtraction, etc. also take fixed cycles. Every instruction is thus executed in a fixed number of clock cycles.

The clock, therefore, controls the generation of all control signals in the computer.

2.2.8 Instruction Format

An instruction is a window through which the internal resources of the computer are made visible to the user. It specifies the type of operation, the location of operands and the location where the result of the operation is to be stored. The location of operands may be some register name, memory location or immediate data. The immediate data is the data value occurring immediately after the operation code, as part of the instruction. The different types of instruction format are shown in Figure 2.9. The operation code specifies the operation to be performed, such as movement of data, arithmetic operation (addition, subtraction, multiplication, division, etc.), logic operation (AND, OR, XOR, etc.) branch operation (unconditional jump, conditional jump, subroutine call, return, etc.).

Operation code	Add. mode	Operand address		
Operation code	Add. mode	Operand 1 address	Operand 2 address	
Operation code	Add. mode	Operand 1 address	Operand 2 address	Result address

Figure 2.9 Instruction format.

The operand address specifies the location of operand. In case of operands located in memory, their addresses may be specified directly or through an indirect way. The Addressing Mode bits specify the operand location and how to determine the address of the operand in memory.

The simplest type of instruction format contains only the Operation Code but no operand address or addressing mode bits. Many operations like NO Operation, Halt etc. have no operands. In many operations like Decimal Adjust Accumulator, Set Carry, Clear Carry, the location of the operand is prefixed and therefore not mentioned.

The single operand instruction format specifies the Operation Code, the Addressing Mode and the Operand Address. The instructions like Increment or Decrement content of register/memory location, Push/Pop to/from stack, etc. come under single operand instructions. There are also types of instructions where the location of the other operand and the result is prefixed. The operations like IN port and OUT port (result location—accumulator), MUL oper., DIV oper. (other operand location and result location—accumulator) come in this category. The other formats are self explanatory.

2.2.9 Addressing Modes

As mentioned before, the addressing mode specifies:

- Where the operand is located
- How to reach the operand location

The operand may be located as:

- Immediate data as part of the instruction
- Data stored in one of the registers
- Data stored in memory

We shall now deal with each of the above cases of addressing modes.

Immediate addressing mode

In this mode, data is specified as part of the instruction. Data byte or word immediately follows the operation code. For example:

MVI B, 05 (in Intel 8085)

(B) □ 05

MOV BX, 97F0H (in Intel 8086)

(BX) □ 97F0H

Register addressing mode

In this addressing mode, the operand is located in one of the registers. The name of the register in coded form may be specified in the operation code, in addressing mode bits or separately as operand code. For example:

ADD AX, BX (in Intel 8086)

(AX) □ (AX) + (BX)

AX and BX are 16-bit registers

ANL A, R3 (in Intel 8051)

(A) □ (A) .AND. (R3)

A and R3 are general purpose 8-bit registers.

Memory addressing modes

When a data byte/word is located in memory, its address needs to be determined to access the data and to perform the specified operation. The address of the operand may be

- specified directly in the instruction.
- given indirectly in the instruction.
- mentioned in the instruction with respect to content of the other register.
- given indirectly in the instruction but with respect to the content of the other register.

These modes are described below.

Direct addressing mode: In this memory addressing mode, the memory operand address is specified as part of the instruction. For example:

LDA 2400H (in Intel 8085)

(A) □ (2400H)

MOV AL, 0FH (in Intel 8086)

(AL) □ (0FH)

OR AL, (2507H) (in Intel 8086)

(AL) □ (AL) .OR. (DS:2507H)

□ (AL) .OR. ((DS) * (10H+2507H))

In Intel 8086, all the addresses are calculated with respect to the segment register contents. DS is the data segment register.

Indirect addressing mode: In this memory addressing mode, the instruction specifies the place (a register or a memory location) where the operand address is stored. Thus if the operand address is stored in X, the instruction would specify as follows: Perform the operation on the operand whose address can be determined using X. X may be a register or a memory location. For example:

MOV M, A (in Intel 8085)

((H) (L)) □ (A)

In Intel 8085, the indirect address is stored in H-L register. The above instructions transfer the contents of A register (Accumulator) to memory location whose address is given in the H-L register pair.

ADD A, @R0 (in Intel 8051)

(A) \square (A) + ((R0))

The contents of the memory location whose address is given in R0 register are ANDed to A register contents and the result is stored back in A register.

Indexed addressing mode: The indexed addressing mode is useful when different elements of a list or an array are to be accessed in loop for performing the same set of operations. In this mode:

- The base of the array is specified in a register called Base register.
- The index or distance of the element from the base of the array to be accessed is specified in a register called Index register.

The sum of the contents of Base register and Index register is the address of the operand in memory.

Let us assume that the ten numbers, X1, X2, ..., X10 are stored in memory in consecutive locations starting from 0200H. The address 0200H is the base of the array, and will be stored in Base register. For the first element X1, the distance from the base, i.e. index, is 0. Thus to access the first element, the Index register will contain 0.

To access X5, the Index register will contain 04.

Thus, the operation in loop may be performed as follows:

Start: Load the Base register with the base location of the array

Load the Index Register with 0

Loop: Perform Operations

 Increment the Index Register

 If the Index Register contents less than or equal to N
 then branch to loop else continue.

This addressing mode is called Base plus Index Indirect Addressing mode. For example:

MOVC @ A + DPTR (in Intel 8051)

(A) \square ((A) + (DPTR))

DPTR contains the base address whereas the register A has the index value.

MOV (BX + DI), AL (in Intel 8086)

((DS)*10H + (BX) + (DI)) \square (AL)

BX is Base register, DI is Index register and DS is Data segment register. All memory address calculations in Intel 8086 use one of the segment registers.

An extension of Base plus Index Indirect Addressing mode is Base plus Index Register Relative Addressing mode, which is useful in case of two-dimensional arrays.

In this case:

- Base register contains the base address of the array.
- Index register contains the increment from base to the row where the data is stored.
- Displacement mentioned in the instruction or in a separate register is the displacement of the element from the start of the row.

Let us assume that the following array of data is stored.

Y00	Y01
Y10	Y11
Y20	Y21

These elements will be stored in memory as Y00, Y01, Y10, Y11, Y20, Y21,

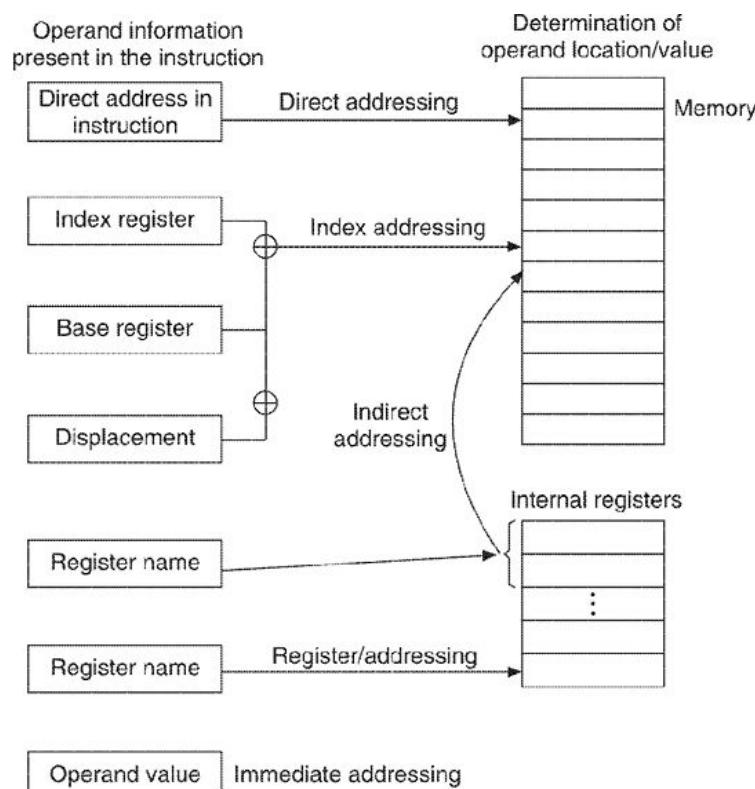


Figure 2.10 Addressing techniques.

To access Y11 the base register will contain the address of Y00, the index register will contain 02, i.e. index of start of row1 from base and the displacement will be 01. For example:

```
MOV (BX + DI + 4), CL (in Intel 8086)
((DS)*10H + (BX) + (DI) + 4) □ (CL)
```

BX is Base register, DI is Index register and 04 is the displacement. DS is Data Segment register.

Figure 2.10 illustrates all of the above addressing modes. There may be many variations, combinations and extensions of these addressing modes, as you will see when studying different microprocessors.

2.2.10 Instruction Set

There are five general groups in which the instructions of a computer may be divided. They are described below.

Data transfer group

This group of instructions deals with the movement of data in byte/word/bit form between

- Register to Register
- Memory to Register
- Register to Memory
- Immediate data to Register
- Immediate data to Memory
- Exchange of data between two registers

Some examples are:

(i) LDA 2000H (in Intel 8085)

Loads data from memory location 2000H to A register.

(ii) MOV BX, CX (in Intel 8086)

Moves the contents of CX register to BX register. The contents of CX register remain unchanged.

(iii) MOV A, # 02 (in Intel 8051)

02 is loaded to A register.

Arithmetic and logic group

The operations performed in this group of instructions relate to the following:

ARITHMETIC OPERATIONS

- Addition of two data bytes/words
- Addition with carry of two data bytes/words
- Subtraction of two data bytes/words
- Subtraction with borrow of two data bytes/words
- Multiplication of two data bytes/words
- Division of two data bytes/words
- Increment of a data byte/word present in register/memory
- Decrement of a data byte/word present in register/memory

Some examples are:

(i) ADD A, R3 (in Intel 8051).

Adds the contents of register R3 to A.

(ii) INC R0 (in Intel 8051)

Increments the register R0.

(iii) MULB R4, R1 (in Intel 8096)

Multiplies the contents of register R1 by R4.

LOGICAL OPERATIONS

- AND, OR, XOR, operations between two data bytes/words present in register/memory
- NOT operation on a data byte/word present in register/memory
- Shift/Rotate (left/right rotation with/without carry operations on data byte/word present in register/memory)

Some examples are:

(i) ANI 0FH (in Intel 8085)

Logically AND the contents of A register with 0FH.

(ii) XOR AL, BL (in Intel 8086)

Logically Exclusive OR the contents of AL and BL registers.

STRING MANIPULATION

- Loading/storing of byte/word string elements
- Movement of byte/word string elements to another memory location
- Comparison of two byte/word string elements.

Some examples are:

(i) MOVSB (in Intel 8086)

Move the string byte from DS:SI to ES:DI in memory

(ii) CMPSW (in Intel 8086).

Compare the string word at DS:S2 with the string word at ES:DI

PROGRAM CONTROL GROUP

This group of instructions includes:

- Conditional/Unconditional branch
- Conditional/Unconditional call to subroutine
- Return from subroutine
- Restart

Some examples are:

(i) JNZ label (in Intel 8085)

Jump to Label if zero flag is not set.

(ii) SJMP Label (in Intel 8096)

Program takes a short jump (-1024 to +1023 bytes) to Label.

(iii) LCALL subroutine Name (in Intel 8051)

Long subroutine call to subroutine Name.

EXECUTION CONTROL INSTRUCTIONS

The instructions in this group relate to:

- No operation
- Halt

- Enabling/Disabling Interrupts
- Initialization of control/function registers for interrupts, on chip timer/counter (if any)

Some examples are:

(i) SIM (in Intel 8085)

Set Interrupt Mask. ACC contents are transferred to Restart Interrupt Mask register

(ii) NOP (in Intel 8085, Intel 8086)

No Operation

(iii) HLT (in Intel 8085, Intel 8086)

Program halts

In addition, there are some instructions which fall under more than one category. Basically two or more instructions are compounded in one instruction in these cases. For example,

DNJ Rn, ‘address’ in Intel 8051, performs two functions:

- Decrements register Rn
- If Rn # 0 then control is transferred to ‘address’

2.3 PROGRAMMING SYSTEM

2.3.1 Machine Language Program

The instructions we have discussed so far are decoded directly by the control unit and executed. For this reason, they are called machine instructions and the language is called machine language. Machine language consists of different binary numbers (bit patterns) for different operations. To perform a particular action by computer or to solve a problem, one would need to write a number of instructions, each performing a step in calculation or action. This combination of instructions is called a program in computer parlance. When the computer was first invented, machine language was the only language available to the users.

As an example of a machine language program, let us consider two numbers stored in memory location 2400(Hex) and 2401(Hex). We need to add these numbers and store the result in memory location 2402(Hex).

Let us assume that 2400(Hex) contains 05 and 2401(Hex) contains 06. In machine language of Intel 8085 microprocessor, the program will look like:

0011	1010	(Code for LDA)	;	Load the content of 2400
0000	0000		;	Hex in Accumulator
0010	0100			
0100	0111	(Code for MOV B, A)	;	Move Accumulator content to B register
0011	1010	(Code for LDA)	;	Load content of 2401 Hex to
0000	0001		;	Accumulator
0010	0100			
1000	0000	(Code for ADD B)	;	Add the content of B register

									;	to Accumulator
0011	0010	(Code for STA)							;	Store Accumulator content in
0010	0100								;	2402 Hex.
0010	0100									

Note that in Intel 8085 machine program the low-order byte is stored first followed by the high-order byte in case of 16-bit numbers. If the program is loaded, starting from memory location 2000 Hex, the memory content will look like:

2000										
(Hex)	0	0	1	1	1	0	1	0		
2001										
(Hex)	0	0	0	0	0	0	0	0		
2002										
(Hex)	0	0	1	0	0	1	0	0		
2003										
(Hex)	0	1	0	0	0	1	1	1		
2004										
(Hex)	0	0	1	1	1	0	1	0		
2005										
(Hex)	0	0	0	0	0	0	0	0		
2006										
(Hex)	0	0	1	0	0	1	0	0		
2007										
(Hex)	1	0	0	0	0	0	0	0		
2008										
(Hex)	0	0	1	1	0	0	1	0		
2009										
(Hex)	0	0	0	0	0	0	1	0		
200A										
(Hex)	0	0	1	0	0	1	0	0		
2400										
(Hex)	0	0	0	0	0	1	0	1		
2401										
(Hex)	0	0	0	0	0	1	1	0		
2402 (Hex)										

The program execution will start at 2000 Hex. For each instruction:

- Instruction will be loaded
- Instruction will be decoded
- Control signals will be generated.

Though the program execution is fast, it is very difficult to write programs in machine language. A single bit error in any instruction will produce undesirable results.

2.3.2 Assembly Language Program

The limitations of machine language led to the development of assembly language. In assembly language, each machine instruction has been replaced by a mnemonic which signifies the operation, e.g. ADD (for addition), SUB (for subtraction), LOAD (for memory read), STORE (for memory write), etc. It became easier for the user to write programs in assembly language rather than in machine language.

The programme in Intel 8085 assembly language for the above example will look like as follows:

```

ORG      2000H (Hex)
LDA      2400H
MOV      B, A
LDA      2401H
ADD      B
STA      2402H
END

```

It is obvious that an assembly language program is easier to understand than the corresponding machine level program.

The assembly language programs are translated to machine level programs using another sub-system called **assembler** as shown in Figure 2.11(a).

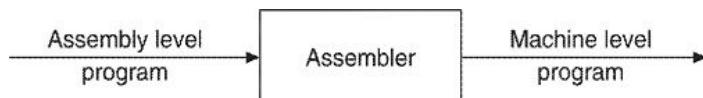


Figure 2.11(a) Programming system (Assembler).

In the memory, a machine level program will look the same as mentioned before and will be executed in the same way. Thus assembler is nothing but a software program which translates assembly programs to their machine level programs. It is the machine language program which is then executed by the computer.

2.3.3 Assembler Directives

There are two instructions in assembly program given above which are not the actual instructions of Intel 8085 microprocessor and as such these are not translated into machine instruction.

1. ORG 2000H indicates to Intel 8085 assembler that the following instructions are to be loaded, starting from memory location 2000 Hex.
2. Similarly END indicates to Intel 8085 assembler that the program has ended and thus no further translation of assembly instructions to machine instructions is required.

Such instructions are known as Pseudo instructions or Assembler Directives. Every assembler will have Pseudo Instructions/directives to help the programmer to communicate the program requirements to the assembler. A common assembler directive is:

Operation—Define

EQU—Define a name

For example

```
X EQU 02
Y EQU 05
```

will define X and Y with 02 and 05 and while translating the assembly program into machine level program, X and Y will be replaced by 02 and 05 respectively.

The other pseudo instructions used by Assembler for the same purpose are, SET and DEF.

Operation—Data

DB—Defines Byte

DW—Defines Word

DDW—Defines Doubleword

For example:

```
N DB 07
```

One byte storage identified with label N and 07 stored in the byte.

```
M DW 400
```

One word (2 bytes) storage identified with label M and 400 (decimal) stored in the word.

```
K DB 04, 05, 06, 09, 10
```

An array with 5 rows, identified with label K and 5 bytes reserved. Values 04, 05, 06, 09 and 10 stored in consecutive locations starting from K.

Operation—Origin

ORG address

The program following the ORG statement will start at the memory location pointed by ‘address’, specified in ORG directive. For example:

```
ORG 1800H
```

```
MOV A,B
```

```
ORG 2400H
```

```
MVI C,08
```

The starting address of MOV A,B will be 1800H whereas MVI C,08 will be loaded starting from location 2400H.

Operation – End

END

It marks the end of assembly language program. Once the END directive occurs, the assembler will not read further instructions and translate them into machine language.

Operation—List

LIST

Provides listing of the assembly language program.

Operation—Define storage

DSB—Define Storage Byte

DSW—Define Storage Word

DDSW—Define Double Storage Word

For example:

TEMP DSB 5

This program will reserve 5 storage bytes in memory and assign the name TEMP to the first location.

PRQ DSW 10

This program will reserve 10 words in memory and assign the first location to PRQ.

It must be noted that the above directives are mostly common in different assemblers. In addition, each assembler will have the same specific directives that reflect the architecture of computer/microprocessor as well as its assembly language.

For example, Intel 8086 microprocessor has SEGMENT, ASSUME, ALIGN to cater for different memory segments like Code segment, Data segment, Extra segment and Stack segment. The specific pseudo instructions will be defined along with the assembly language of the microprocessor in different chapters.

2.3.4 Compilers

With the growth of computers, the problems that the users wanted to solve on computer became more and more complex. The assembly language programs became very lengthy and unmanageable. Extensive programming efforts were needed to write even a simple program. This limited the scope and utility of computers. However, later with the development of high-level languages like BASIC, FORTRAN, ALGOL, C etc. the program writing became easier. To execute the high-level language programs on a computer, it is necessary to translate them into machine level language which can then be directly executed by computers. This task is achieved by a compiler which is shown in Figure 2.11(b).

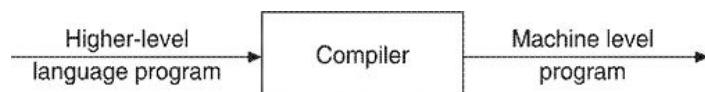


Figure 2.11(b) Programming system (Compiler).

Compilers and assemblers are software programs used by the computer to convert the programs written in high-level and assembly language respectively into machine language programs. It is clear that a separate compiler is necessary for each language. A computer supporting three high-level languages will need three separate compilers.

2.3.5 Operating Systems

A supervisory package (resident with the computer) is necessary for the operation of the software packages with the user program. This supervisory package is called **operating system**. The operating system also allocates computer resources to the user program. In different environments, the operating system plays different roles and is quite complex.

2.4 WHAT IS MICROPROCESSOR?

The achievements in the area of microelectronics have brought a revolution in the field of computers. The advent of integrated circuit in 1957 was the turning point.

Since then, lot of research has been pursued to accommodate maximum amount of circuitry on a single IC chip. Whereas in 1970, an IC chip could contain 3000 transistors, in 1980 the number of transistors on one single IC chip increased to 70,000. It was possible to put more than 80,000 transistors on a single chip using NMOS technology by 1990. Simultaneously, CMOS technology came up as a competitor to NMOS technology. CMOS offers low power requirement, which is vital for many applications.

Due to these advancements in microelectronics technology, it became possible to put ALU and Control Unit on a single IC chip. This IC chip is called **microprocessor**. The microprocessor is considered as a major revolution in the field of computers. It has also been made possible to integrate ALU, Control Unit and Memory on a single IC chip. This IC chip is called **microcomputer**. The microelectronics technology is continually advancing and consequently the microprocessors are becoming more powerful. Now we are soon to enter the world of nano electronics, which will offer many more advantages.

Figure 2.12(a) explains the concept of microprocessor and microcomputer in von Neumann Computer organization shown in Figure 2.1. We shall now discuss the general facilities, which should be offered by any microprocessor.

The microprocessor chip has to be interfaced to memory and I/O units to work as a computer for any application. Figures 2.12(b) and 2.12(c) show the microprocessor and microcomputer chips connected to memory and I/O units. Even though the microcomputer chips contain memory on chip, options for interfacing external memory are provided in most of the chips.

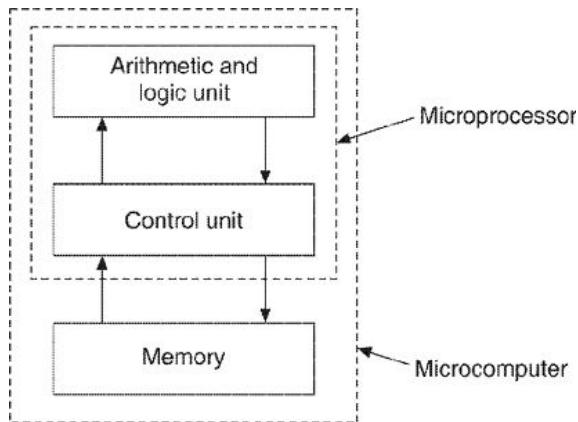


Figure 2.12(a) Microprocessor and microcomputer—the basic concepts.

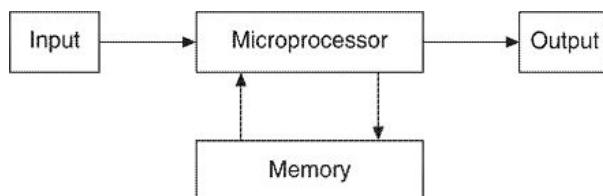


Figure 2.12(b) Microprocessor interfaced to memory and I/O units.

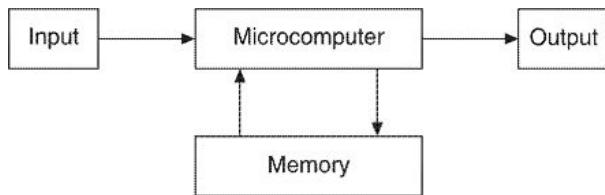


Figure 2.12(c) Microcomputer interfaced to memory and I/O units.

Clearly the microprocessor should be able to accept and send data from and to memory and I/O units. To read a data from memory, the microprocessor should generate

- address of the location, and
- read control signal for memory read.

The microprocessor should also be able to accept data from memory. For interfacing the microprocessor to any input or output device the same information would be required. Thus address, data and control signals are required to interface the microprocessor to memory and I/O devices.

2.5 ADDRESS BUS, DATA BUS, AND CONTROL BUS

A bus is defined as a path over which digital information is transferred from any of the several sources to any of the several destinations. It can be a dielectric medium or a set of physical wires carrying the signal. A microprocessor chip has thus an address bus, a data bus, and a control bus associated with it (Figure 2.13). The data bus is bi-directional as the microprocessor accepts as well as sends data, while the address bus is unidirectional since address locations are sent by microprocessor to memory and I/O devices.

The control signals fall into the following two categories

- Control signals like Read from memory or I/O or Write to memory or I/O are output by microprocessor to memory or I/O devices.
- Control signals like Interrupt Request, DMA request, Reset, Halt, etc. are sent to microprocessor by I/O devices.

Thus, the control bus is bi-directional. However, unlike the data bus, any particular line will have information flow in one direction only.

A careful examination would reveal that considering microprocessor as ALU + Control Unit, Figure 2.12(b) has been reorganized and presented in Figure 2.13.

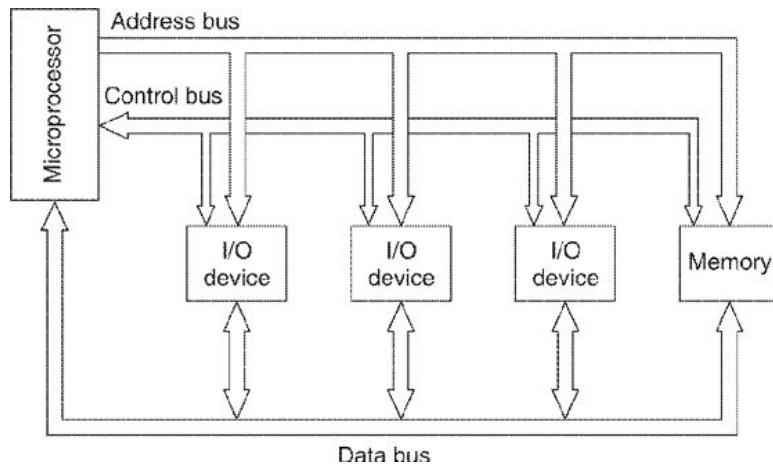


Figure 2.13 Microprocessor buses.

2.6 TRISTATE BUS

Figure 2.14 shows three data transmitters X, Y, Z connected to a single bus through the output of gates x, y, and z. It is obvious that only one of the transmitters should transmit at any particular instant.

Conventional TTL gates can have only two outputs, i.e. 0 or 1. Thus if the gates x, y, z are TTL gates, there will always be a signal present at their outputs and the information at the output point will be garbled. What we require is that X, Y should disconnect when Z is transmitting information to bus line, Y and Z should disconnect when X is transmitting and X and Z should disconnect when Y is transmitting. To fulfil this requirement, the gates x, y, and z should have the following three states

- a logic 0 state
- a logic 1 state
- a state in which output is effectively disconnected from rest of the devices.

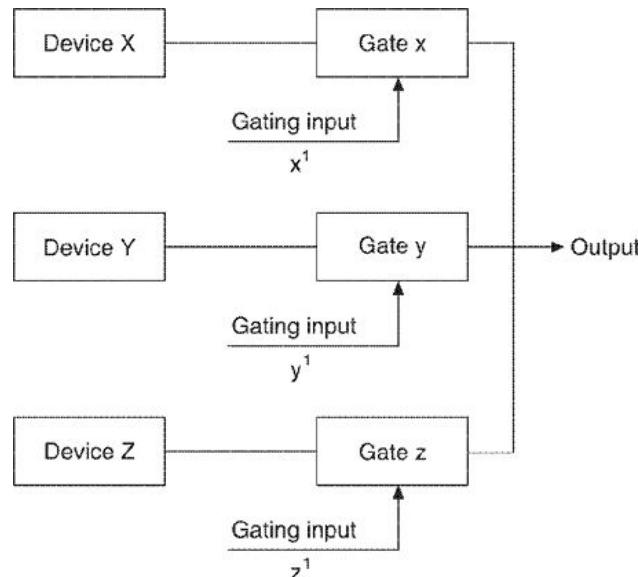


Figure 2.14 Tristating—an example.

The third state is called high impedance state. These devices have an extra input called enable/disable, which permits the logic gate to

- either behave normally or else
- disconnect the output of the gate from rest of the output.

The enable/disable is shown as gating input x^1 , y^1 and z^1 for gates x, y and z in Figure 2.14.

The symbol for tristate logic gate is shown in Figure 2.15.

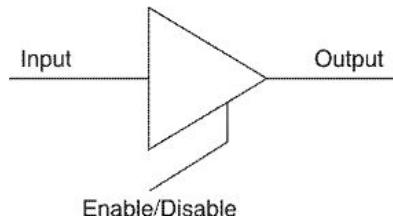


Figure 2.15 Tristate logic gate.

Let us now discuss the general features common to all microprocessors. Apart from data bus and address bus, many other signals such as clock, direct memory access, interrupts, and control and status signals are required. Without these signals, a microprocessor will not be useful in many of the applications.

2.7 CLOCK GENERATION

Early microprocessors needed clock input to be given externally, i.e. an extra clock generator chip was necessary. The clock generator chip had two pins between which a crystal or an *RC* circuit could be connected for the generation of basic frequency desired (Figure 2.16). However, microprocessors, that were designed after 1978 (Intel 8085, M6809, etc.) had the clock generator circuit embedded in the microprocessor chip. A crystal or *RC* network can be connected between the two pins (X1 and X2). To synchronize other I/O devices with the microprocessor, it outputs the clock as ‘Clock Out’ as well. In a multi-microprocessor environment, this ‘Clock Out’ can be connected to other microprocessors directly (Figure 2.17).

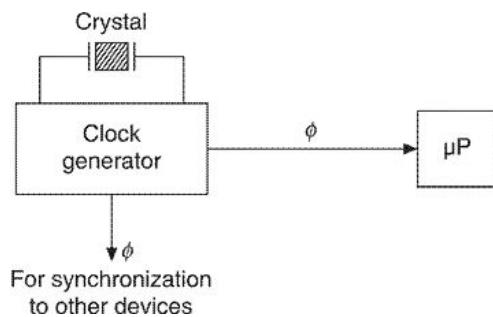


Figure 2.16 Microprocessor with clock generator.

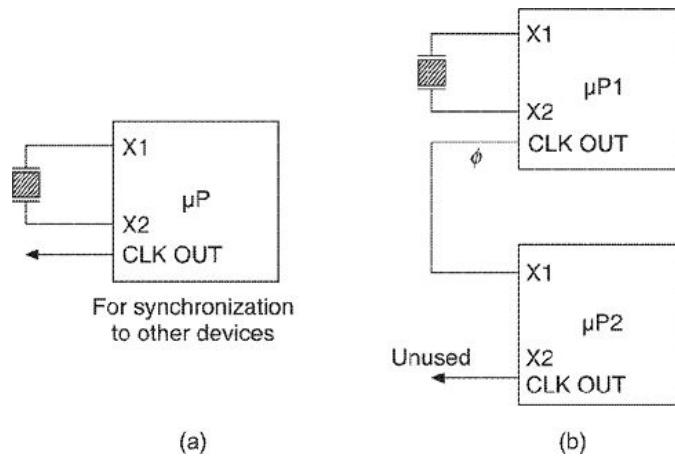


Figure 2.17 Microprocessor with on-chip clock generator.

2.8 CONNECTING MICROPROCESSOR TO I/O DEVICES

2.8.1 I/O Mapped I/O Interface

Figure 2.18 shows the connection between microprocessor, I/O devices and memory. I/O devices are identified by port numbers and memory locations by addresses. The memory read/write operations and I/O read/write operations are performed by different software instructions. Whether the read/write operations are being performed on memory or I/O, or in other words whether the information on address and data lines is meant for a memory location or an I/O device—this identification is done by separate signals. Thus, when read from an I/O device instruction is executed, the I/O signal is ON and the address on the address bus is decoded as the port number and an I/O device is selected.

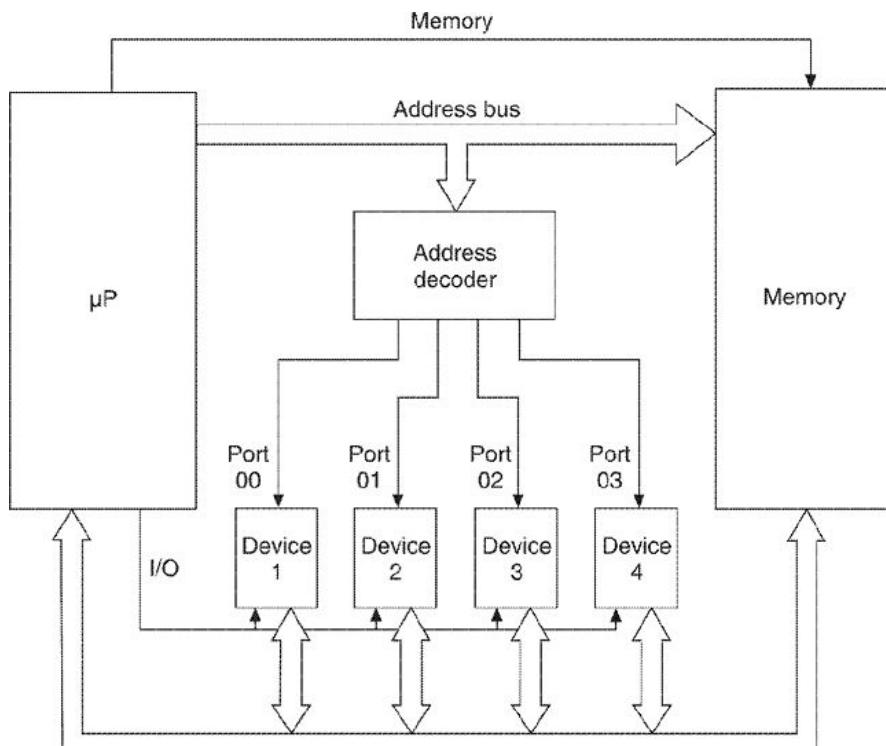


Figure 2.18 I/O mapped I/O interface.

In case of read/write from memory, the MEMORY signal is ON and a particular location of memory is selected. In Figure 2.18, the port numbers of devices 1, 2, 3 and 4 are 00, 01, 02, and 03 respectively. Because of separate memory and I/O signals, there is no confusion between device address, i.e. port number and memory address. This is called I/O mapped I/O interface since I/O devices are treated separately from memory.

2.8.2 Memory Mapped I/O Interface

Let us now see Figure 2.19, which is identical to Figure 2.18 except that signals I/O and MEMORY are not present. Now when a read memory instruction is executed, there is no MEMORY signal to indicate that the address bus contains the memory location address. If this memory location address is the same as that of a port number of an I/O device, an I/O device will also get selected together with the memory read operation being performed. Thus, there will be confusion between memory location and I/O device having the same address and port number.

In Figure 2.19, when one wishes to read from device 1 (port no. 00) memory location 00 will also get selected. However, if we sacrifice some memory locations for the sake of I/O devices, this problem would not arise. It means that I/O addresses (port numbers) and memory addresses will not be the same. In Figure 2.19 if the memory starts from address 04 onwards, then there would not be any problem. But what advantage is achieved due to this sacrifice of memory locations?

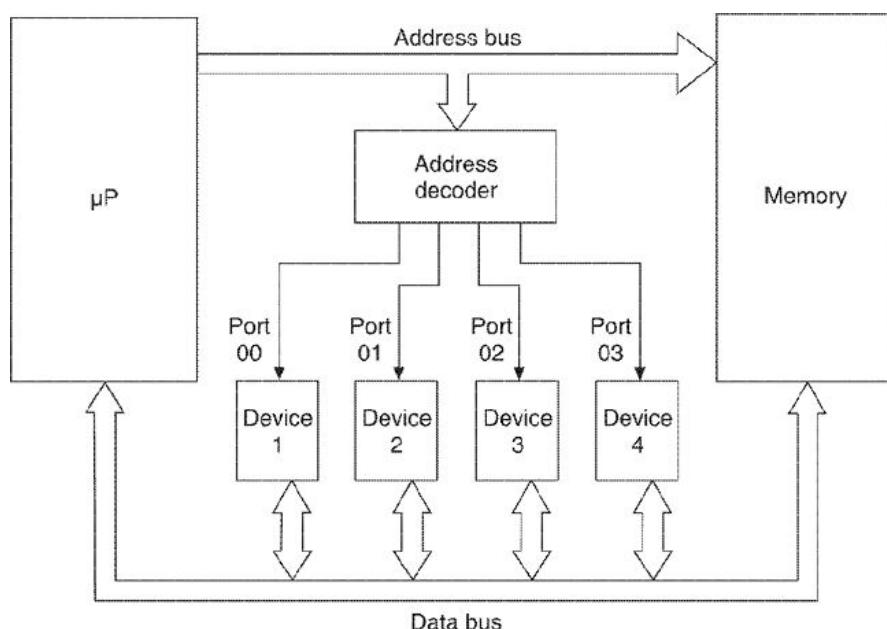


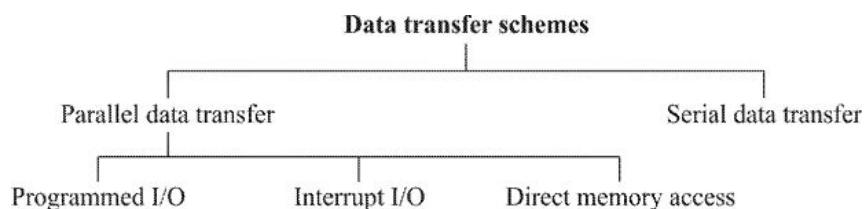
Figure 2.19 Memory mapped I/O.

The memory read/write instructions are quite versatile and powerful in general. If a microprocessor has more than one register apart from ACC (Intel 8085 has 6 registers other than ACC), then memory operations can be performed using any of these registers. These instructions automatically become valid for I/O devices if connected in this fashion. The I/O read/write instructions in I/O mapped I/O normally require the transfer from I/O port to accumulator. The same data is then subsequently transferred to the other register, thus wasting one instruction.

The memory read/write instructions employ various powerful addressing modes like indexed, indirect, base register addressing, etc. which is not true in the case of I/O read/write instructions. Thus, more compact and more efficient handling of I/O devices can be achieved if they are interfaced to microprocessor in this way. Since an I/O device is treated as memory location (identified by memory address), this interface is called memory mapped I/O.

2.9 DATA TRANSFER SCHEMES

Data transfer schemes depend heavily on the environment (on-line or off-line processing), type of I/O device (capable of parallel or serial data transfer, synchronous or asynchronous) and the application. Data transfer schemes may be categorized as shown below.



2.9.1 Parallel Data Transfer

Programmed I/O

In programmed I/O, the data transfer is controlled by the user program being executed. Depending on the type of the device, data transfer may be synchronous or asynchronous. Synchronous data transfer is used when the I/O device matches in speed with the microprocessor. The microprocessor issues the read/write instruction addressing the device whenever data transfer is required. The actual data transfer takes place in one clock cycle.

When the I/O device and microprocessor speed do not match, i.e. when the I/O device is slower than the microprocessor, asynchronous data transfer is used. In this mode of data transfer, microprocessor checks the status of the device. If the device is not ready, the microprocessor continuously checks the status of the device till it becomes ready. The data transfer instruction is then issued by the microprocessor (Figure 2.20).

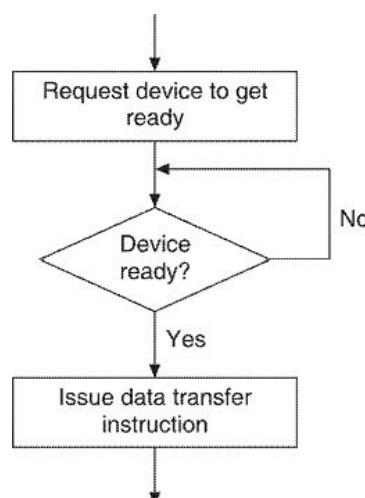


Figure 2.20 Asynchronous data transfer.

Interrupt I/O

The data transfer scheme in Figure 2.19 is quite inefficient, since the microprocessor is kept busy for the slower I/O device. The remedy to this problem is to allow the microprocessor to do its job when the device is getting ready and when the device is ready, the microprocessor can transfer the data. This can be achieved through interrupt.

Interrupt is the facility provided by the microprocessor to the outside environment by which the attention of the microprocessor can be diverted to do some higher priority job. The interrupts are used for varied purposes in different environments. Microprocessor can be interrupted to initiate data transfer, to execute control sequence to control large power plants, or to check the status of a process at any particular instant.

Clearly, the microprocessor should scan the signal on the interrupt pin during every machine cycle. When the interrupt signal is present, it should suspend the current job. The current status of the suspended job should be stored so that the microprocessor can restart the suspended job from the same point. The stack is used to store the status of the suspended job. The microprocessor services the interrupt request by executing an Interrupt Service Routine. The interrupt operation is explained in Figure 2.21.

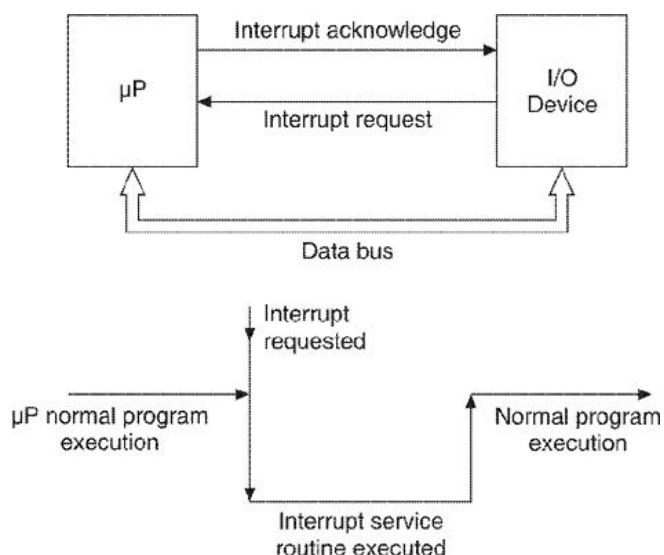


Figure 2.21 Interrupt operation.

In interrupt I/O (Figure 2.22) data transfer scheme, the microprocessor initiates the device. When the device is ready to transmit or receive the data, it sends an interrupt request signal. The microprocessor completes the current instruction execution and then suspends the current job, saves the address and current status in stack and then executes the Interrupt Service Routine. On completion of data transfer, the microprocessor retrieves the status of the suspended job and restarts the operation. Following is the operation sequence for interrupt operation.

- Normal program execution by microprocessor.
- Microprocessor initiates the device through a code/signal (e.g. start convert signal to initiate ADC conversion).

- (c) The device when ready to send the data sends an interrupt signal on one of the interrupt pins.
- (d) Microprocessor checks the validity of interrupt request by checking whether
- the interrupt system is enabled.
 - the particular interrupt is not disabled.
 - any higher priority interrupt is not pending or being processed.
- (e) If an interrupt request is valid, the microprocessor
- completes the current instruction execution.
 - saves the status register and Program Counter(PC) in stack.
 - issues interrupt acknowledgement signal.
 - determines the address of interrupt servicing routine and stores the starting address in PC. The program thus branches to Interrupt Servicing Routine.

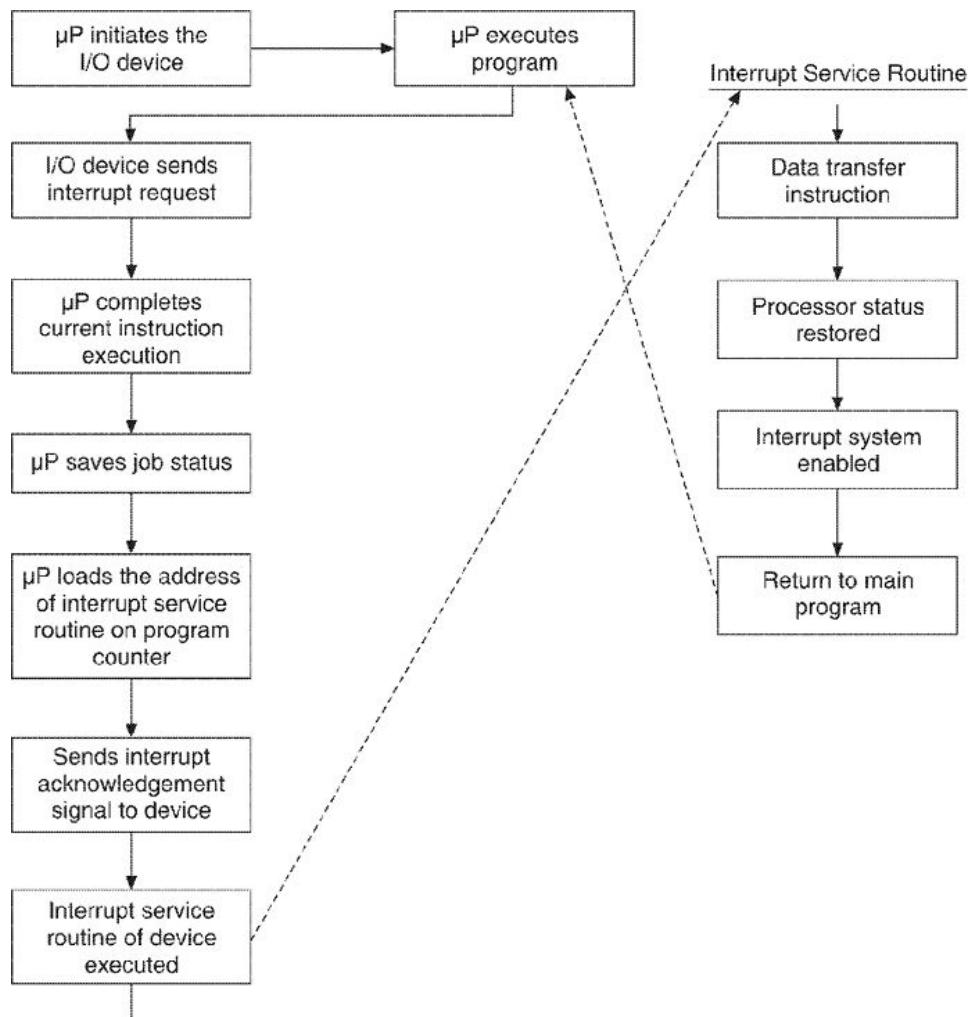


Figure 2.22 Interrupt data transfer.

The last instruction of Interrupt Servicing Routine is ‘Return’. When this instruction is executed, the PC and the Status Register are loaded back from stack. Thus normal program execution is resumed.

Direct memory access

In programmed I/O and interrupt I/O, data is transferred to the memory through the accumulator. This process is quite uneconomical for bulk data transfer, when the I/O

device matches the speed of the microprocessor. In such cases the device is allowed to transfer the data directly to memory, bypassing the microprocessor. The I/O device requests the microprocessor for Direct Memory Access (DMA) by sending a signal on a special pin. The microprocessor disconnects itself from memory and I/O device by tristating the address, data and control buses and acknowledges the device by sending DMA acknowledgement signal. The I/O device performs the data transfer. On completion of data transfer, the I/O device intimates the microprocessor by withdrawing the DMA request (Figure 2.23).

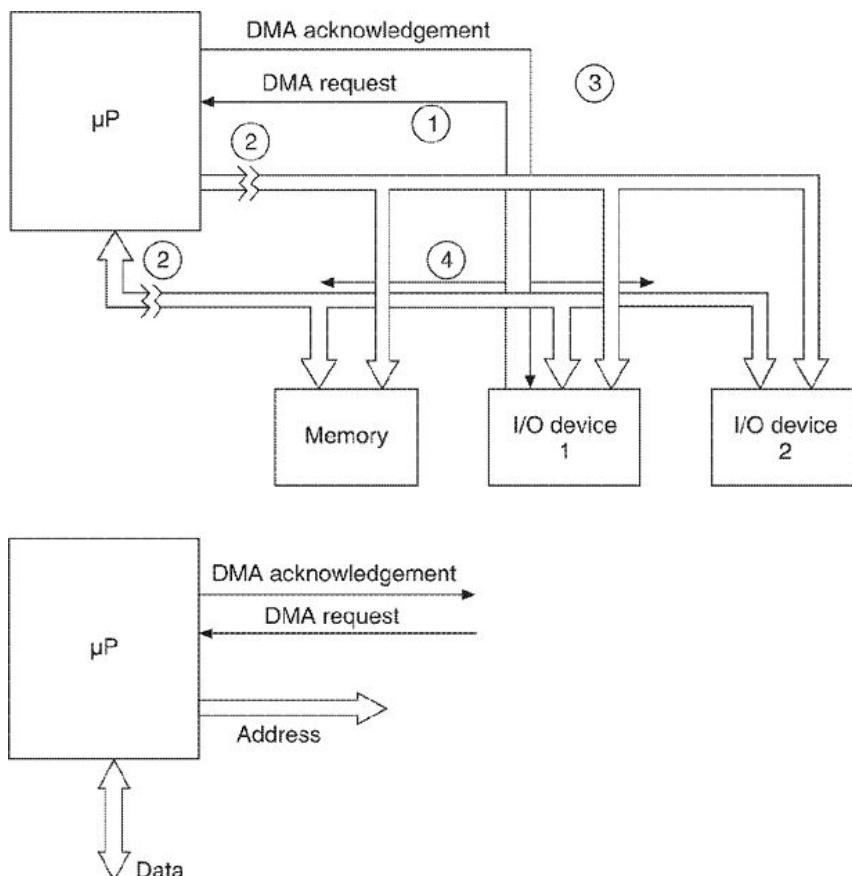


Figure 2.23 Direct memory access.

Following is the operation sequence in case of Direct Memory Access.

1. The microprocessor checks for DMA request signal once in each machine cycle.
2. The I/O device sends signal on DMA request pin.
3. The microprocessor tristates address, data and control buses.
4. The microprocessor sends acknowledgement signal to I/O device on DMA acknowledgement pin.
5. The I/O device uses the bus system to perform data transfer operation on memory.
6. On completion of data transfer, the I/O device withdraws DMA request signal.

7. The microprocessor continuously checks the DMA request signal. When the signal is withdrawn, the microprocessor resumes the control of buses and resumes normal operation.

2.9.2 Serial Data Transfer

Some devices like CRT receive and transmit data in serial mode. In addition, in many applications, a number of microprocessor systems are connected to each other in network fashion to form a distributed microprocessor system. The data transfer between two processors is in serial mode. The data is transferred bit by bit on a single line. This minimizes the number of interconnecting wires. The microprocessor providing serial data transfer facility will have two pins for input and output of serial data and special software instructions to affect the data transfer.

2.10 ARCHITECTURAL ADVANCEMENTS OF MICROPROCESSORS

A number of advancements that had taken place in the computer have migrated to microprocessor field with the continual advancement of microelectronics technology. The concepts of multitasking, pipelining, and multiprocessing are already there in the latest microprocessors. The pipelined microprocessors have a number of smaller independent units connected in series like a pipeline. Each unit is able to perform its tasks independently and the system as a whole is able to give much more throughput than the single microprocessor. Multitasking provides the environment in which the microprocessor can execute multiple tasks simultaneously by cycle stealing. The concept of virtual memory increases the memory capacity beyond the physical memory space possible through width of address bus. The Memory Management Unity (MMU) is now integrated on microprocessor chips. Thus microprocessors of the 2000s are equivalent to supercomputers of 1990s and this process of growth is going to continue in the following years as well.

We shall now describe some of the concepts like pipelining, cache memory and virtual memory.

2.10.1 Pipelining

Pipelining in computer is used to enhance the execution speed. A number of processing elements or processors are used in pipelining. To explain the concept, let us assume that a task T is to be executed on a computer. This task T can be further divided into N sub-tasks as T(1) to T(N). These sub-tasks are independent of each other and the output of sub-task I is the input of sub-task I + 1 and thus the output of sub-task T(N) is the final output of task T. If we have N processors, P(1) to P(N), which are connected in the same way, i.e. the output of processor P(1) is the input to processor P(2) and so on (Figure 2.24) and if all sub-tasks take equal time, this connection will then achieve N times enhancement in the execution speed.

Let us consider the example of four processing elements P(1) to P(4). The task T has to be subdivided into four sub-tasks T(1) to T(4). Each processor takes 1 clock cycle to execute its sub-task as depicted below.

Cycle No	P(1)	P(2)	P(3)	P(4)
	□	□	□	□

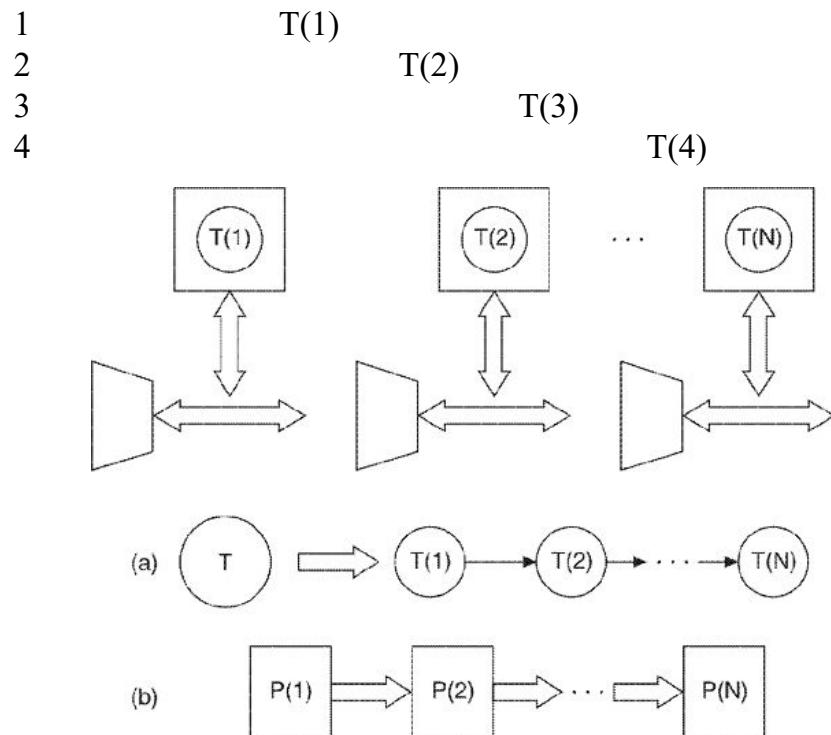


Figure 2.24 Independent task allocation.

Thus a single task T will take 4 clock cycles which is same as a non-pipelined processor. Therefore, a conclusion can be drawn that, for a single task there is no advantage derived from a pipelined processor. However, let us consider an example in which there is continuous flow of tasks T(1), T(2), T(3), T(4) ... each of which can be divided into four sub-tasks [e.g., task T(1) will have sub-tasks T1(1), T2(1), T3(1) and T4(1)] as before. Now the pipeline will look like

Cycle	P1	P2	P3	P4
1	T1(1)			
2	T1(2)	T2(1)		
3	T1(3)	T2(2)	T3(1)	
4	T1(4)	T2(3)	T3(2)	T4(1)
5	T1(5)	T2(4)	T3(3)	T4(2)
6	T1(6)	T2(5)	T3(4)	T4(3)

Now

- The task T(1) consisting of sub-tasks T1(1), T2(1), T3(1) and T4(1) will get completed at the end of cycle 4.
- The task T(2) consisting of sub-tasks T1(2), T2(2), T3(2) and T4(2) will get completed at the end of cycle 5.
- Similarly task T(3) will get completed at the end of cycle 6.

Thus all tasks after task T(1) take one cycle to get completed. Thus for subsequent tasks, 4 times enhancement in speed has been achieved. If the number of tasks are very large, the total speed enhancement (including task T(1)) approximates to 4.

This technique has been used to increase the instruction's execution speed in number of computers. The technique is called 'Instruction Pipeline'. An instruction execution contains the following four independent sub-tasks.

- (a) Instruction Fetch
- (b) Instruction Decode
- (c) Operand Fetch
- (d) Execute

If four hardware modules or processing elements are designed to execute these four sub-tasks, 4 times enhancement in the execution speed is possible.

This however is the maximum theoretical enhancement possible.

Unfortunately, not all instructions are independent. If instruction I_2 has to work on the result of the previous instruction I_1 , i.e. the result of instruction I_1 is the operand for instruction I_2 , then the operand fetch for I_2 has to wait for I_1 to get completed. Control logic inserts stall or wasted clock cycles into pipeline until such dependencies are resolved.

Another problem frequently encountered is that relating to branch instructions. As an example, if instruction I_2 is a branch instruction, the execution of instructions I_3 and I_4 in the pipeline becomes meaningless as program will branch to a new location. In such cases, the entire pipeline must be flushed. The problem increases if the number of stages in the pipeline are more (e.g. Intel Pentium 4 has a 20-stage pipeline).

The problem could be alleviated by incorporating branch prediction. Suppose as part of instruction decode, it is possible to predict

- whether a branch will take place (in case of condition branch).
- the location to which the program control will be transferred.

then by the time the instruction is executed, the pipeline can be flushed and execution of instruction from the new address may be initiated.

However, a poor branch prediction may further exacerbate the problem.

Intel 8086, 80186, 80286 and 80386 have a 2-stage pipeline, i.e. Bus Interface Unit (Instruction Fetch) and Execution Unit (Instruction Execution).

Intel 80486 has a 5-stage instruction pipeline whereas Pentium processor has a 20-stage instruction pipeline with branch prediction.

2.10.2 Cache Memory

Increasing memory throughput will increase the execution speed of the processor. To achieve this, one or more fast buffers named **cache** between the processor and the main memory are provided. The cache memory has cycle time compatible with the processor speed. This facilitates the data and instruction to be supplied to the processor fast enough for processing. The working of a cache memory system is explained in Figure 2.25.

The cache memory consists of a few kilobytes of high speed Static RAM (SRAM), whereas the main memory consists of a few Megabytes to Gigabytes of slower but cheaper Dynamic RAM (DRAM). Now let us consider the operation.

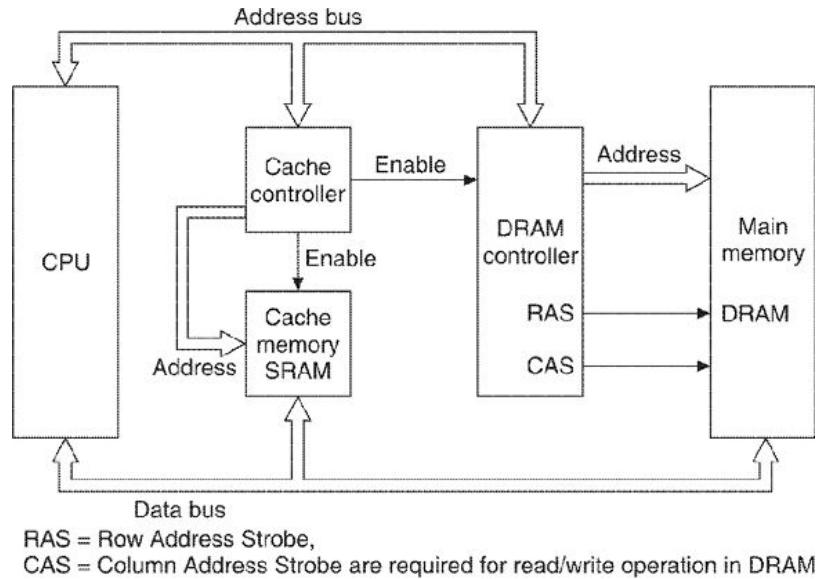


Figure 2.25 Cache memory—principle of operation.

- When the CPU wants to read a byte or word, it outputs address on the Address Bus.
- The cache controller checks whether the required contents are available in cache memory.
- If the addressed byte/word is available in cache memory, the cache controller enables the cache memory to output the addressed byte/word on Data Bus.
- If the addressed byte/word is not present in cache memory, the cache controller enables the DRAM controller.
- The DRAM controller sends the address to main memory to get the byte/word. Since DRAM is slower, this access will require some wait states to be inserted.
- The byte/word which is read is transferred to CPU as well as to cache memory. If CPU needs this byte/word again, it can be accessed without any wait state.

The percentage of accesses where the CPU finds the desired byte/word in cache is called hit rate.

For write operation, if the cache controller finds that the addressed byte/word is present in the cache memory then the controller will write the new byte/word to the cache memory without any wait state. It intimates the CPU that the write operation is complete. The cache controller then writes the byte/word to main memory.

The above write operation methodology is called the Posted Write Through Method. The write to main memory is transparent to the CPU unless main memory is involved in the previous write operation.

To keep track of which main memory locations are currently present in cache memory, the cache controller uses a cache directory. Each location in the cache is represented by an entry in the directory. The exact format of the cache directory depends on the cache scheme used. The basic cache schemes are—Direct mapped

cache, Two way set associative cache and Fully associative cache. We shall not describe these here. Interested readers may consult references 1 and 4 on this topic.

Modern processors have incorporated both specialized caches as well as cache hierarchy. Pipelined CPUs access memory from multiple points in pipeline, i.e. different processing elements in pipeline access memory for different information. Different physical caches for each of these points are used in the architecture. The idea is to allocate one physical resource (cache) to service one point in pipeline. Thus three separate caches—Instruction, TLB (Translation Look Aside Buffer) and Data are incorporated to serve Instruction Fetch, Virtual to physical address translation and Data fetch.

A **victim cache** is used to hold blocks evicted from a CPU cache due to conflict or capacity miss. It is put between the main cache and its refill path. The blocks that were evicted from main cache on a miss are stored in victim cache. The technique reduces the penalty incurred by cache on a miss. The microprocessors AMD K7 and K8 have large secondary cache as victim cache to avoid duplicate storage of contents of large primary cache.

Trace cache is a specialized cache incorporated to store traces of instructions that have already been fetched. A trace cache stores instructions either after decode or after their retirement. Generally, instructions are added to trace caches in groups representing either basic blocks or dynamic instruction traces. A basic block contains a group of non-branching instructions ending with a branch. On the other hand, a dynamic trace consists of instructions whose results are actually used, and eliminates instructions taken from following branches since they are not executed. The trace cache mechanism is used to increase the instruction fetch bandwidth. Intel microprocessor **Pentium 4** uses trace caches to store already decoded micro-operations so that the instructions, if needed next time, can be taken from trace caches without going for decode.

The simulation studies of cache behaviour vis-à-vis cache size have indicated that capacity miss (i.e. misses which a cache of given size will have regardless of its associativity or block size) falls steeply between 32 KB and 64 KB. Thus a CPU designer will like to keep the cache size around 64 KB.

Another consideration is the trade-off between cache latency and hit rate. Large caches have better hit rates but are slower. To overcome this, many computers use multiple levels of cache memory. Figure 2.26 shows three levels of cache memory system. In a multilevel cache

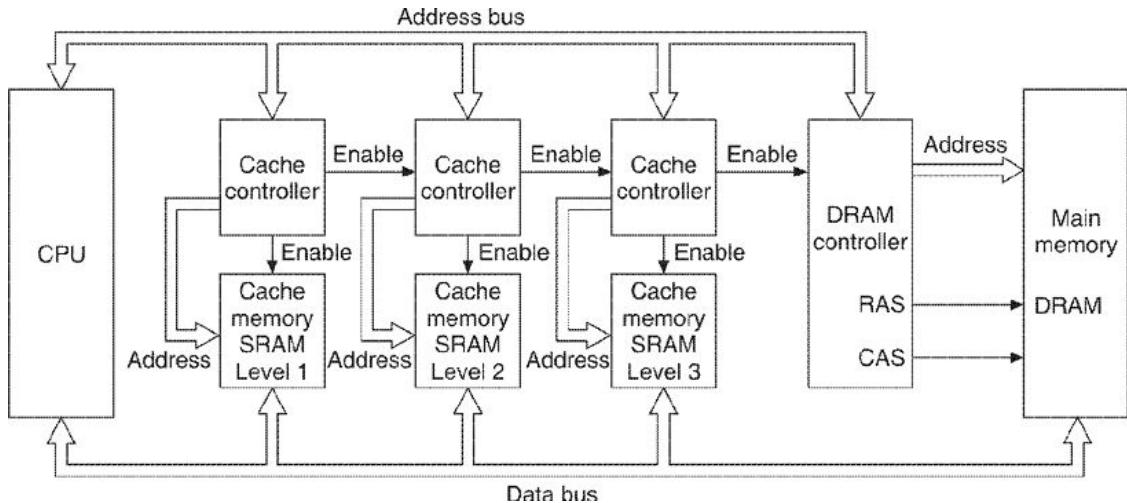


Figure 2.26 Multilevel cache system.

system, the small but fast cache is used first, followed by large and slower caches which are ultimately followed by even slower main memory.

As explained in the Figure 2.26, multilevel caches operate by checking the smallest level 1 cache first. If the addressed byte/word is found, i.e. it hits, then the processor proceeds with high speed, otherwise the next higher cache is checked, and so on. At the end, main memory is checked. Multilevel caches can be either **inclusive** or **exclusive**. In an inclusive cache design the data in L1 cache will also be in L2 cache. Intel microprocessors Pentium 2, 3 and 4 have the inclusive multilevel cache system.

In exclusive caches, data is present either in L1 or L2 caches. AMD microprocessor Athlon follows the exclusive multilevel cache system. The advantage of exclusive caches is that they store more data. When L1 misses and L2 hits on an access, the hitting cache line in the L2 is exchanged with a line in L1.

The advantage of inclusive cache is that when external devices or other processors in a multiprocessor system wish to remove a cache line from the processor, the processor needs to check only the L2 cache. In the other designs both L2 and L1 cache will require to be checked.

Though considered very small, but at the extreme case, the 6-byte instruction queue of Intel 8086 Processor can be considered an example of on-chip cache. Other examples include Pentium 2, 3, 4 using the inclusive multilevel cache system and AMD Athelon 64 (whose core design is known as K8) using the exclusive multilevel cache.

Itanium II processor uses 6 MB unified level 3 cache on-chip. IBM Power 4 series has 256 MB level 3 cache off-chip, shared among several processors.

2.10.3 Memory Management

The simplest model of memory management in computers in the beginning was Linear Addressing or Linear Memory organization. Here, only a single program occupied the complete memory space. The systems designer/programmer made a plan to use the memory space according to application and then developed the software. The earliest microprocessor like Intel 8080, Z80, and Intel 8085 used this concept.

Even embedded processors like Intel 8051, Intel 8096 use this model as the processors are dedicated to specific applications.

With the passage of time, as computers grew in hardware complexity, they were required to be used in timesharing multitasking environments. This involved allocating a fixed amount of time to programs of different users. Relocatability of programs became the main criteria. In order to achieve relocatability, segmented addressing was used. In this mode of addressing the addressing memory space is divided into a number of segments. The segment registers hold the memory address of the beginning of the segment. All addresses in programs are in relation to the memory addresses stored in segment registers. Figure 2.27 shows that the complete memory segment of 64K bytes is divided into 3 segments, A, B and C.

- A Segment Reg = 0 0 0 0 H
- B Segment Reg = X X X X H
- C Segment Reg = Y Y Y Y H

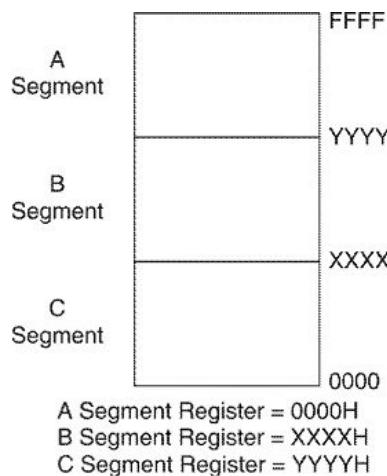


Figure 2.27 Segmented memory system.

Segment Registers A, B and C store the starting address, i.e. Base Address 0000H, XXXXH and YYYYH of segments in memory. Now each memory address, whether for instruction or for data is defined as

$$\text{Memory Address} = \text{Base Address} + \text{Offset}$$

The offset for any information stored in memory is the distance in memory location from the start of the segment.

Now consider that a program P is loaded in segment A starting from address WWW. Relocating this program in another segment will require changing only the base address. Thus in multitasking systems, different user programs could be located as per the availability of memory space. Intel 8086 incorporated a segmented memory system with four segments—Code Segment, Data Segment, Extra Segment, and Stack Segment.

2.10.4 Virtual Memory System

As computer applications grew in complexity and consequently computer programs became complex and lengthy, it became evident that physical memory size would become a bottleneck. The virtual memory system was evolved as a compromise in

which the complete program was divided into fixed/variable size pages and stored in hard disk. The main memory was also divided into pages of the same size. The basic idea is that pages can be swapped between the disk and the main memory as and when needed. This task is performed by the operating system. Thus, for a programmer, the main memory size is much more than the physical memory available which is virtually correct.

The software designer decides the virtual address space of his program and data. The virtual address space is divided into pages. The physical memory is also divided into corresponding units of the same size called frames. Figure 2.28 shows an example where page size is 4K bytes, physical memory is 64K bytes and the virtual memory is 256K bytes.

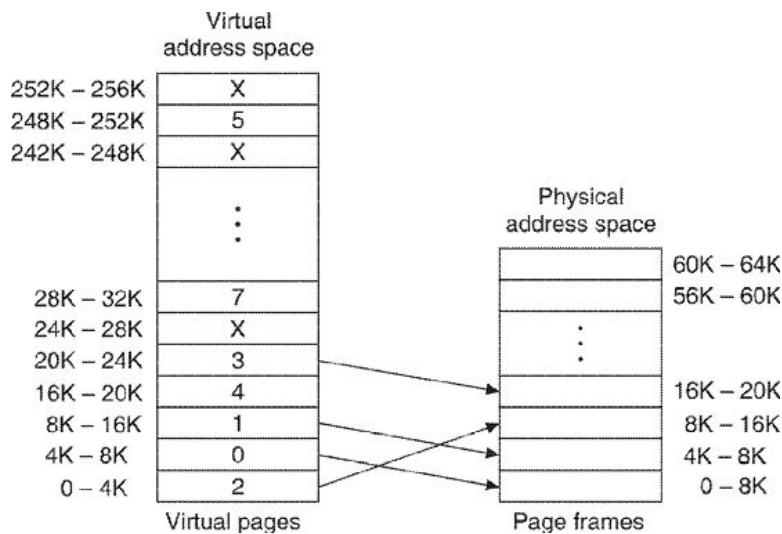


Figure 2.28 Mapping of virtual address space and physical address space.

$$\text{Number of pages in virtual space} = 64$$

$$\text{Number of page frames in physical memory} = 16$$

Let us assume that a virtual address of 8292 (8K + 100) is generated (for instruction fetch or memory read/write as a result of instruction execution) by the CPU. This occurs at virtual page No. 3. Here 100 is the offset within the page. This virtual address is mapped to page frame No.1 in physical address space which starts at 4K. Thus the physical address will be $4K + \text{offset} = 4096 + 100 = 4196$.

The task of translation of virtual address to physical address is carried out by the Memory Management Unit (MMU). The MMU is a hardware device, functionally positioned between the CPU and actual memory. However, many processors like Intel 80286, 80386, 80486, Pentium and Motorola MC 68031 and MC 68040 have MMU integrated on-chip.

The memory management unit uses the page table to map virtual pages to page frames. Figure 2.29 shows the example of how virtual address 8292 is translated to physical address using page table. Since page size is 4K, i.e. 2^{12} , 12 bits of the address whether physical or virtual are assigned to offset whereas the upper 6 bits in case of virtual address will be assigned to virtual page number. Similarly the upper 4 bits will be assigned to a page frame number. For each page, the page table keeps information on the present/absent, i.e. whether the page is present in main memory or not.

The physical address may be loaded to bus. The page frame numbers in page table are the most significant bits of the physical memory address. If the page is present in memory, memory fetch is effected, otherwise the MMU raises a page fault interrupt. The operating system selects a page frame where the required page may be loaded from the disk. The contents of the selected page frame are saved into disk, and the referenced page is fetched from the disk and is placed in the selected page frame. The operating system changes the page table and restarts the instruction that caused the page fault.

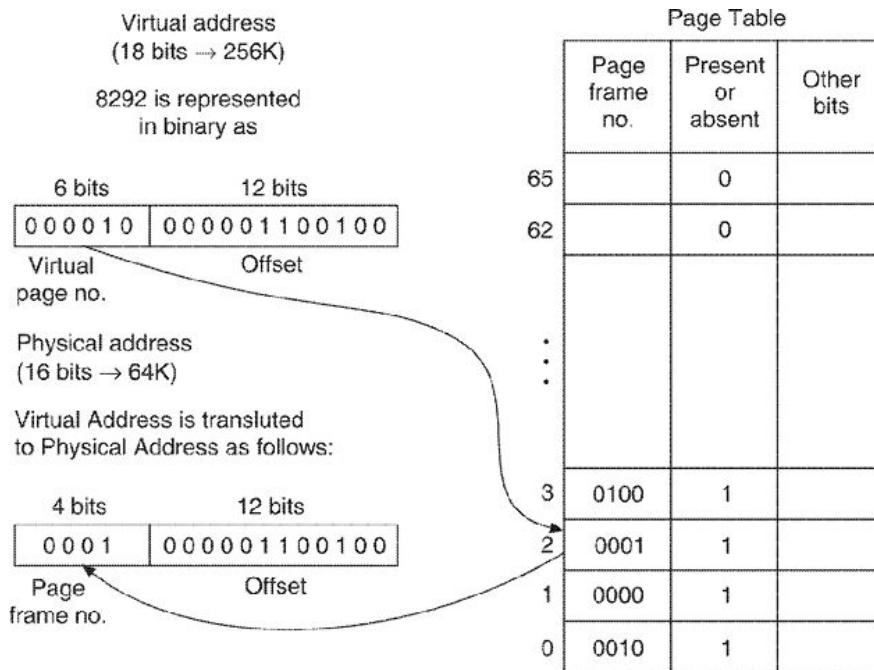


Figure 2.29 Translation of virtual address to physical address.

In addition to the Present/Absent bit, the page frame may contain the following information:

Dirty bit: If bit = 1, the page in the memory has been modified. While page swapping between pages in disk and memory, the dirty bit is examined. If dirty bit = 0, the page in physical memory has not been modified, and thus need not be written back to memory. A new page can be written over this page.

Referenced bit: If bit = 1, the program has made reference to the page, meaning that the page may be currently in use. The operating system normally likes to select a page frame which has not been referenced for page replacement.

Protection bits: These bits specify whether the data in the page can be read/written/ executed.

The mapping from virtual to physical address must be done for every memory reference. Every instruction fetch requires a memory reference. Also, many instructions have a memory operand. Therefore, the mapping must be extremely fast lest it becomes the bottleneck.

For fast operations, a number of page table design alternatives like Multi Level Page Tables, Translation Lookaside Buffers (TLBs) etc. have been suggested. Translation Lookaside Buffers is a fast look-up table to correlate the virtual page number (through key) to the page frame number. The TLB is put as hardware cache in

microprocessor to make the mapping very fast. The Intel Pentium Processor uses 32-bit virtual address with 4K page size.

The MMUs in Intel 80286, Intel 80386 and Intel 80486 manage segment-based virtual memory through the Segment Selector and Descriptor Table. Each address in Intel 80286 has the following two components.

- 16-bit segment selector
- 16-bit offset.

Using the segment selector, the MMU accesses the descriptor for the desired segment in the Descriptor Table. The descriptor contains the base address of the segment, the privilege level of the segment and some other control bits. The selector has 14 bits, meaning that the total number of entries in Descriptor Table (Figure 2.30) and therefore the total number of segments is 16384. Since offset is represented in 16 bits, the total number of memory location, i.e. the size of each segment is 64K. Thus the total virtual space available in Intel 80286 is 1 Gigabyte, as compared to physical memory of 16 Megabyte (MMU has 24 address lines).

You would notice that the basic concept is the same, as described earlier. The descriptor table is basically the page table and the segment selector is the virtual page address.

It is often opined that virtual memory concept is the extension of cache memory, a concept where disk has replaced the main memory and main memory has replaced the cache memory.

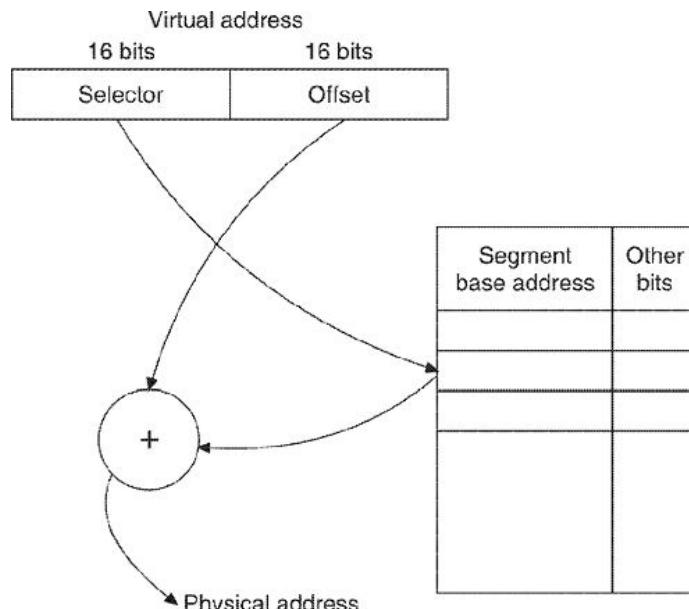


Figure 2.30 Segment-based virtual memory in Intel 80286, 80386 and 80486.

In the following sections, we shall briefly discuss the evolution of microprocessors.

2.11 EVOLUTION OF MICROPROCESSORS

Typical of early microprocessors were the Intel 4004, Rockwell PPSK, Burroughs Mini-d and Fairchild PPS-25, which were all calculator-based chips with emphasis on arithmetic operations. These had surprisingly good internal facilities (such as on-chip registers), but very little emphasis was given to speed. The P-Channel MOS logic at that time was considerably slow in itself and the microprocessors were given a very small package (e.g. 16 pins), which required multiplexing of data streams.

2.11.1 8-Bit Microprocessors

The developments in 8-bit microprocessors occurred due to transition of microelectronics technology from LSI to VLSI technology which is based on N-channel metal oxide semiconductor work. With the high mobility of negatively-charged carriers (electrons) in N-channel MOS, higher logic speeds and greater packaging densities than those in P-channel MOS were achieved.

Performance was improved by using a separate data and address bus with enough address bits for large amount of memory. On-chip registers addressed by the register addressing mode helped to speed up the programs. Instruction sets became more sophisticated with good arithmetic, data transfer and control facilities. The interrupt facility became standard.

With N-MOS, the microprocessors were able to drive the rest of the systems without extra TTL chips, thus reducing the components count. Led by the Intel 8080, every 8-bit microprocessor has its own peculiar characteristics.

Intel 8080, 8085: The Intel 8080 microprocessor (Figure 2.31) represented a dramatic improvement over the 8-bit P-MOS Intel 8008 microprocessor. The Intel 8080 has a separate data and address bus but control signals are multiplexed on the data bus, requiring external latches to extract them. There are several on-chip registers but very few instructions can address the main memory directly—addressing being mostly register indirect. There are separate set of instructions for register, memory and input/output. The speed of execution is quite high. There are useful interrupt and subroutine facilities. It requires an external clock generator circuit and three supply voltages. The architecture reflected a switch to a parallel bus organization though the system interface was still not clear. Since Intel 8080 emerged earlier than other microprocessors, it became very popular and was taken as the industry standard.

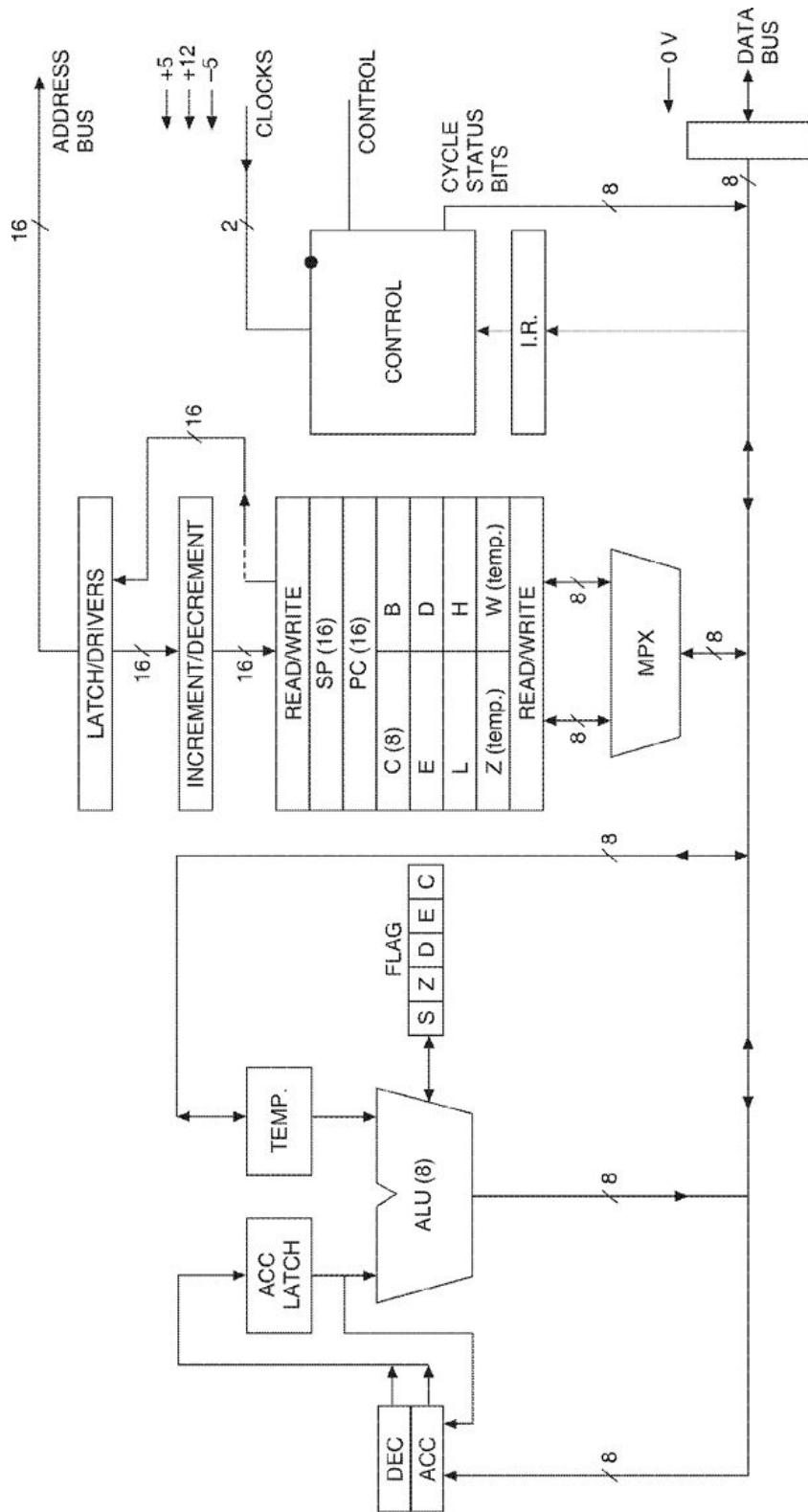


Figure 2.31 Intel 8080 architecture.

The Intel 8085 improved upon the Intel 8080 for it needed only one +5 V supply instead of three different voltages. The clock generator is integrated on-chip and the control lines need not be demultiplexed from the data bus. There are multiple interrupts provided on-chip. Yet the instruction set is almost unchanged and only a little extra speed is offered. On-chip serial input/output facility is present in addition.

It has therefore been aimed to make a system easier and cheaper to build rather than to give higher processing performance.

Zilog 80: The Z 80 microprocessor was designed by the same team who created the Intel 8080. It has twice as many CPU registers as the Intel 8080 and many more instructions. Its advantages over the Intel 8080 are similar to those of the Intel 8085. The instruction set has more addressing modes which facilitate the development of shorter programs of greater speed. The designers have tried to make the instruction, which appear most often in the program, take up the least space and those which are executed most often the fastest.

Motorola 6800, 6802 and 6809: The approach adopted by Motorola for the development of their 8-bit microprocessor was quite different. The microprocessor had two accumulators but no general purpose data register on-chip. A parallel data bus concept was also adopted. All I/O devices were interfaced in memory mapped I/O. Good addressing modes (direct, page, indexed) and single level addressing (memory and I/O read/write use same instructions due to memory mapped I/O) made the program easy to learn. The fastest execution time (2 microseconds) was similar to the Intel 8080, but only a single low-level clock and one +5 V supply voltage was required.

Later versions of 6800 (68A00, 68B00), are much faster than the earlier versions because of the extensive changes in technology. The fastest instruction executes in 1 microsecond (ms).

The 6802 has an on-chip RAM of 128 bytes but is slower. Otherwise, it is similar to 6800 with a clock oscillator on-chip.

Although an 8-bit machine, 6809 (Figure 2.32) has many 16-bit characteristics. It has been made upwards compatible with 6800. The CPU architecture, registers, and instruction set are extensions of 6800 format. The two accumulators of 6809 can be used together for 16-bit operations. There are four index registers (unlike only one in 6800) of which two are also used as stack pointers (U is the upper stack, S is for subroutine/interrupt stack).

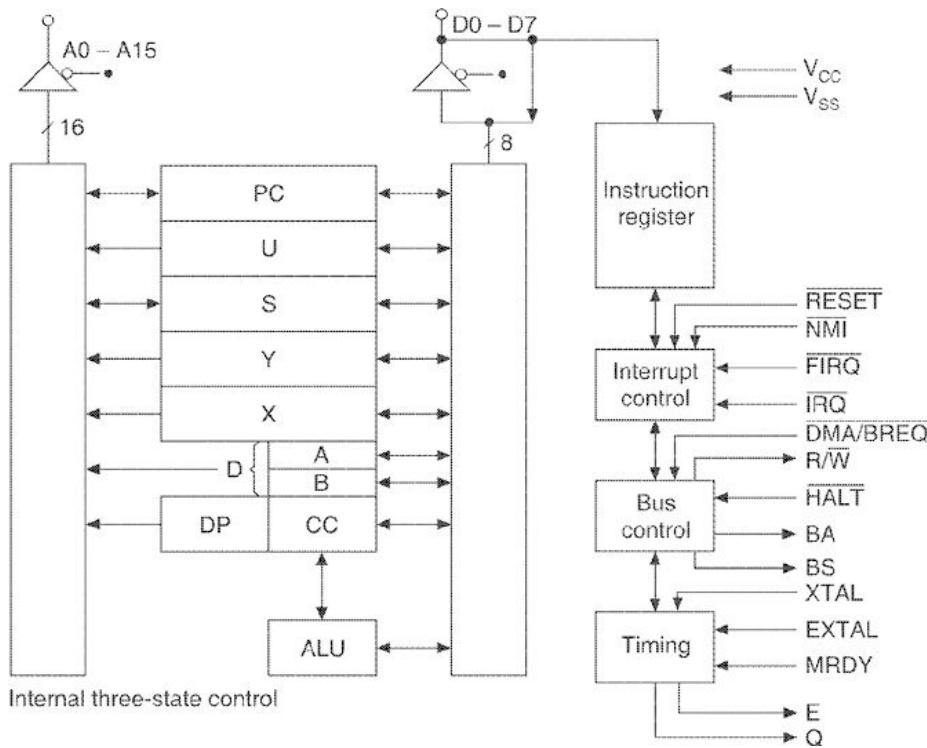


Figure 2.32 MC 6809 block diagram.

There is a page register (DP register) for short branch instructions. The addressing modes are numerous and useful and include variations of direct, indexed, indirect indexed, etc. There is an 8-bit multiply instruction and there are 16-bit add subtract, store and compare instructions. The 6809, therefore, represents an attempt to offer an enhanced 8-bit microprocessor which is economical to use.

2.11.2 16-Bit Microprocessors

The advent of the 16-bit microprocessor was marked with certain new concepts like pipelining, virtual memory management, etc. Powerful addressing modes were evolved. Multiply and divide instructions were also introduced. The era of the 16-bit microprocessor began in 1974 with the introduction of PACE chip by National Semiconductor, and CP 1600 by General Instruments. Several powerful microprocessors have been developed since then.

Intel 8086, 80186 and 80286: The Intel 8086 (Figure 2.33) was the first of the new breed of high performance 16-bit microprocessors. The HMOS technology allowed over 28,000 transistors to be used in the design and gave it a high speed. Memory components in HMOS are also very fast, down to 100 ns access time. It achieves its speed without needing fast memory components or even a separate data and address bus. The CPU consists of two parts, namely Execution Unit and Bus Interface Unit. These parts act independently to achieve high speed. The Bus Interface Unit maintains a queue of six instructions for Execution Unit. The memory is divided into four segments of 64 Kbytes each. The 20-bit address bus allows 1 million bytes of memory. The use of four segment registers (one for each segment) allows the program modules to be placed anywhere in the memory. Intel 8086 also has two 16-bit pointers, two index registers and four 16-bit general purpose registers.

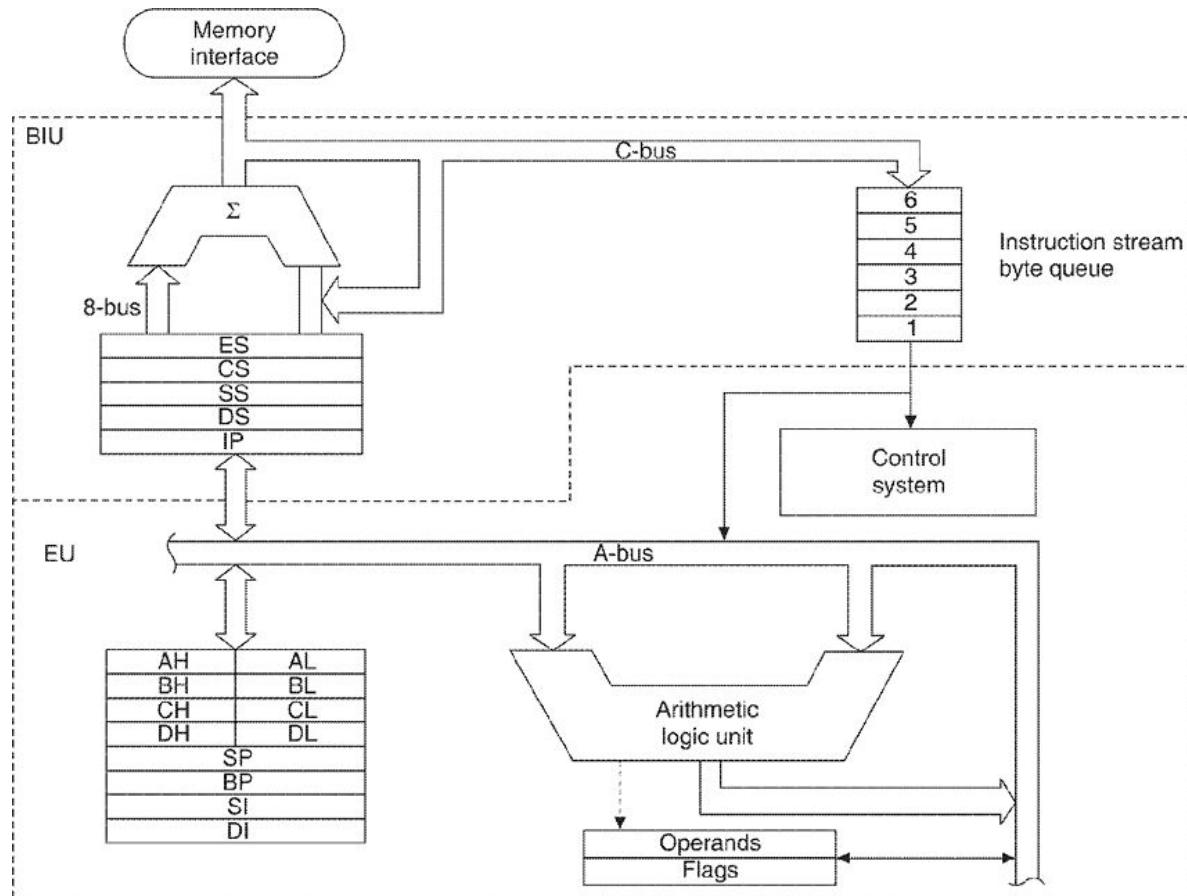


Figure 2.33 Intel 8086 block diagram.

Powerful addressing modes and instruction set have been evolved. The microprocessor uses two address instructions, and operations may take place between two registers, register and memory, register and immediate, or memory and immediate. Memory addresses can be direct, indirect (via base or index register) or indexed (via base or index register). The processor can be operated in minimum or maximum mode by wiring a pin to +5 V or ground.

The maximum mode allows the multiprocessor environment. The Intel 8086 instruction set is upward compatible to Intel 8080 and Intel 8085 with the exception of RIM and SIM instructions. This compatibility is at the source level only.

The Intel 8088 processor was introduced later than Intel 8086. The internal structure of Intel 8088 microprocessor is the same as its predecessor. The difference, however, lies in data bus, which is 8-bit wide in the case of Intel 8088. The Intel 8088 was introduced for systems, which were originally designed around Intel 8080 or Intel 8085 and needed upward compatibility. The Intel 8088 architecture is shown in Figure 2.34.

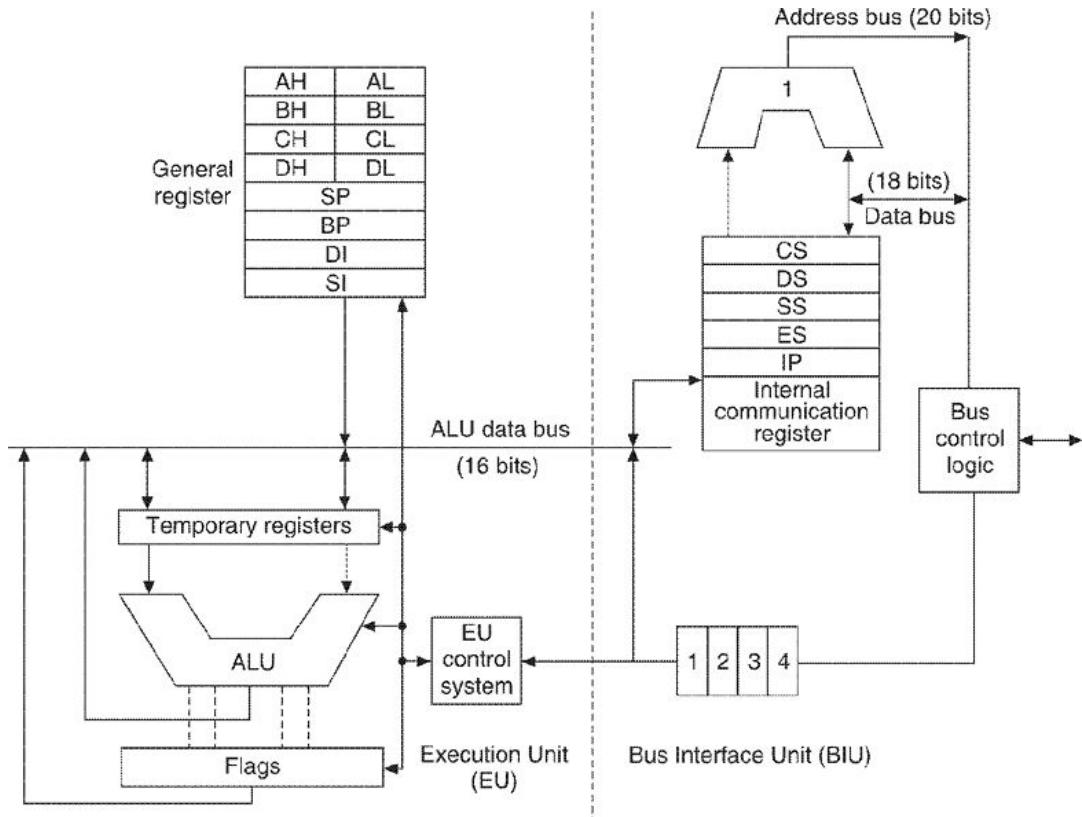


Figure 2.34 Intel 8088 elementary block diagram.

The Intel 80186 (Figure 2.35), introduced in 1982, offers twice the performance of standard Intel 8086 and offers 12 additional instructions. Integrated on the chip are clock generator, DMA controller with two independent channels, three programmable 16-bit timers, Intel 8086 CPU (8 MHz version), etc. The multi-CPU configuration can be achieved through HOLD and HLDA.

The Intel 80286, also introduced in 1982, offers still higher performance, up to 6 times that of Intel 8086. It has on-chip 10 MHz processor (8086), memory management unit with four-level memory protection and support for virtual memory and operating system. It supports 16M bytes physical and 1 Gbytes virtual memory. The Intel 80286 uses a superset of Intel 80186 instruction set with sixteen additional instructions. This processor is specially designed for multiuser and multitasking systems.

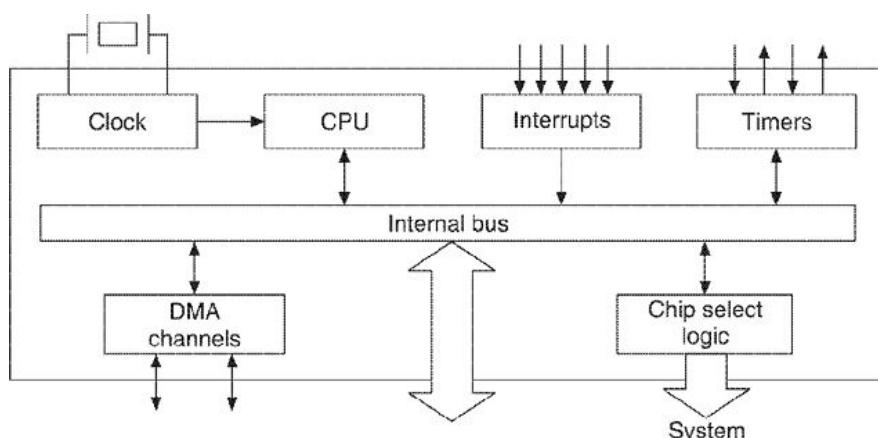


Figure 2.35 Intel 80186 architecture.

Zilog 8000: By using VLSI techniques, Zilog have managed to pack an extremely powerful 16-bit microprocessor on a single NMOS silicon chip. Two basic versions of Z8000 were introduced. Z8001 is a segmented addressing system and it allows access to 23-bit address bus and permits up to 8 million 16-bit words or 16 Mbytes of memory to be used. Z8002 is non-segmented version with 64 Kbytes memory.

It is possible to use Z8000 in multiprocessor environment. Z8000 does not have a dedicated register for use as the accumulator. Instead, it uses a bank of sixteen general purpose 16-bit registers, any of which may be used as accumulator. Memory in the Z8000 system may be divided into areas for system and user and also into separate data and program areas which are all defined by status control lines from the processor. The main addressing modes provided by Z8000 are register, indirect register, direct, immediate, indexed, relative, base address and base indexed. There are 110 basic instruction types, which may be executed on various modes to give over 400 different types of operations.

Motorola 68000: The MC68000 (Figure 2.36) is quite different from other 16-bit processors in many respects. Though it is a 16-bit microprocessor, it has mostly 32-bit wide data organization and a flexible array, which gives it a very high processing throughput. Hardware multiplication and division logic are included on the chip. This further increases the processing speed in complex calculations. Twenty-three address bits provide 16 Mbytes of direct addressable memory space. No dedicated accumulator has been provided; instead, it uses a bank of eight 32-bit general purpose registers D0–D7, of which any register can be used as accumulator. The system is able to handle 32-bit operations with ease.

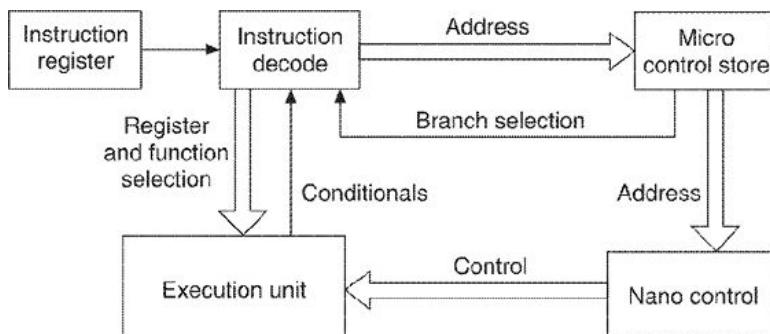


Figure 2.36 Motorola 68000 architecture.

Fourteen different basic addressing modes are provided on the MC68000. There are 56 basic instruction types in the MC68000 instruction set. Combined with the addressing modes, they form a powerful instruction repertoire.

2.11.3 32-Bit Microprocessors

The era of 32-bit microprocessor began in 1981 with the introduction of iAPX 432. In 1980, IBM implemented IBM 370 CPU on a single chip but did not offer it commercially. Other 32-bit processors are Belmac-32A microprocessor from Bell Labs, the 32-bit CPU chip from Hewlett Packard, Intel iAPX 386 series, Motorola 68020 and 68030, etc.

Intel iAPX 386: Intel 80386 is a 32-bit member (Figure 2.37) of iAPX 86 family. It is software compatible to Intel 8086, Intel 8088, Intel 80186 and Intel 80286. New

concepts like caching, pipelining are provided along with high performance bus, and high-speed execution unit. The Intel 80386 provides two to three times the performance of Intel 80286. It has pipelined architecture with parallel fetching, decoding, execution and address translation inside the CPU. It provides full 32-bit architecture and internal implementation including 32-bit register file, instruction queue, address translation unit, address bus and data bus. It has hardware

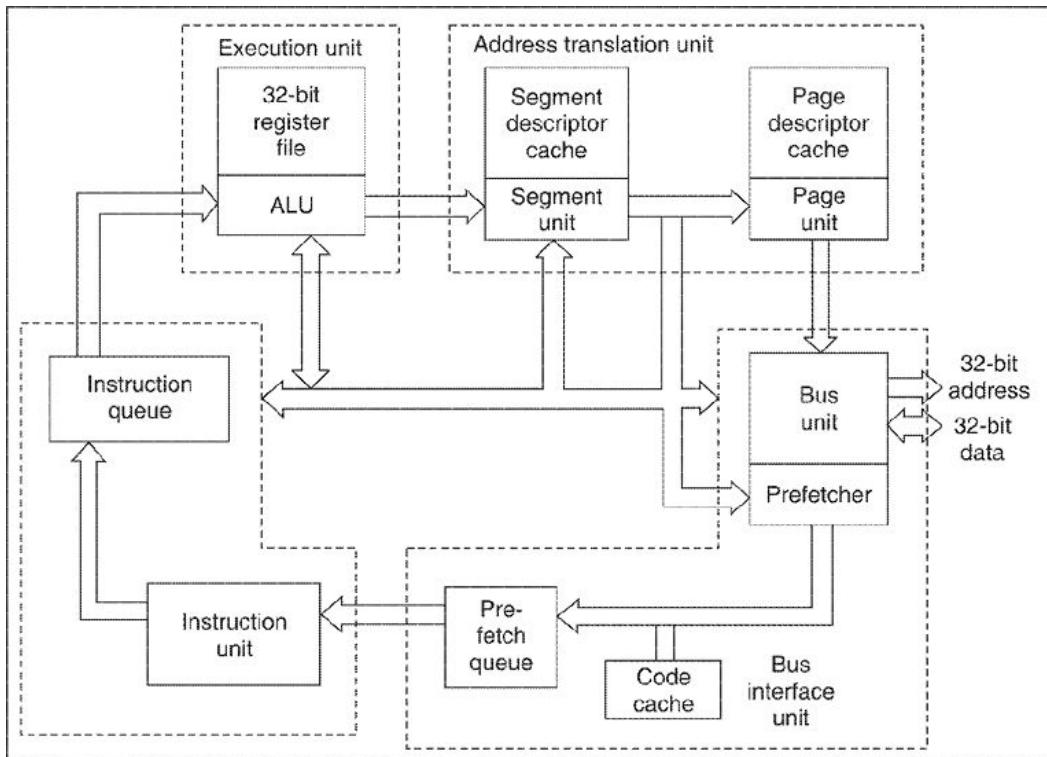


Figure 2.37 iAPX 386 pipeline.

supported multitasking and virtual memory support. The physical memory up to 4 Gbytes can be addressed, whereas the virtual memory up to 64 Tbytes can be addressed per task. It has hardware-enforced protection up to four levels to provide protection of sensitive code data within a task. The general purpose registers of Intel 80386 support 32-bit data and addressing. They also provide for 8-bit and 16-bit compact addressing.

Motorola 68020, 68030: Motorola 68020 (Figure 2.38) is a 32-bit virtual memory processor. It has fast on-chip instruction cache to improve execution speed and bus bandwidth. It is object code compatible to 68000. The pipelined architecture has high degree of internal parallelism, which allows multiple instructions to be executed concurrently. The processor has sixteen 32-bit data and address registers, and supports 18 addressing modes and 7 data types. Four Gbytes memory can be directly interfaced. The clock frequency can be 12.5, 16.57, 20 or 25 MHz.

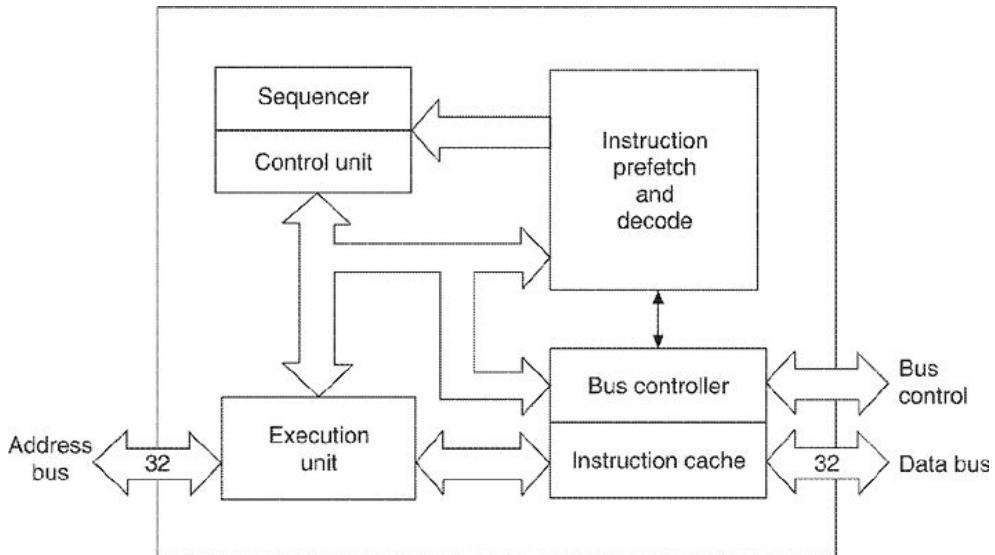


Figure 2.38 Motorola 68020 block diagram.

Motorola 68030 (Figure 2.39) is a second-generation 32-bit enhanced microprocessor based on 68020 core. It is object code compatible with 68020. The paged memory management unit translates addresses in parallel with instruction execution. The processor contains 256 bytes instruction cache and 256 bytes data cache. The clock frequency can be 16.67 MHz or 20 MHz.

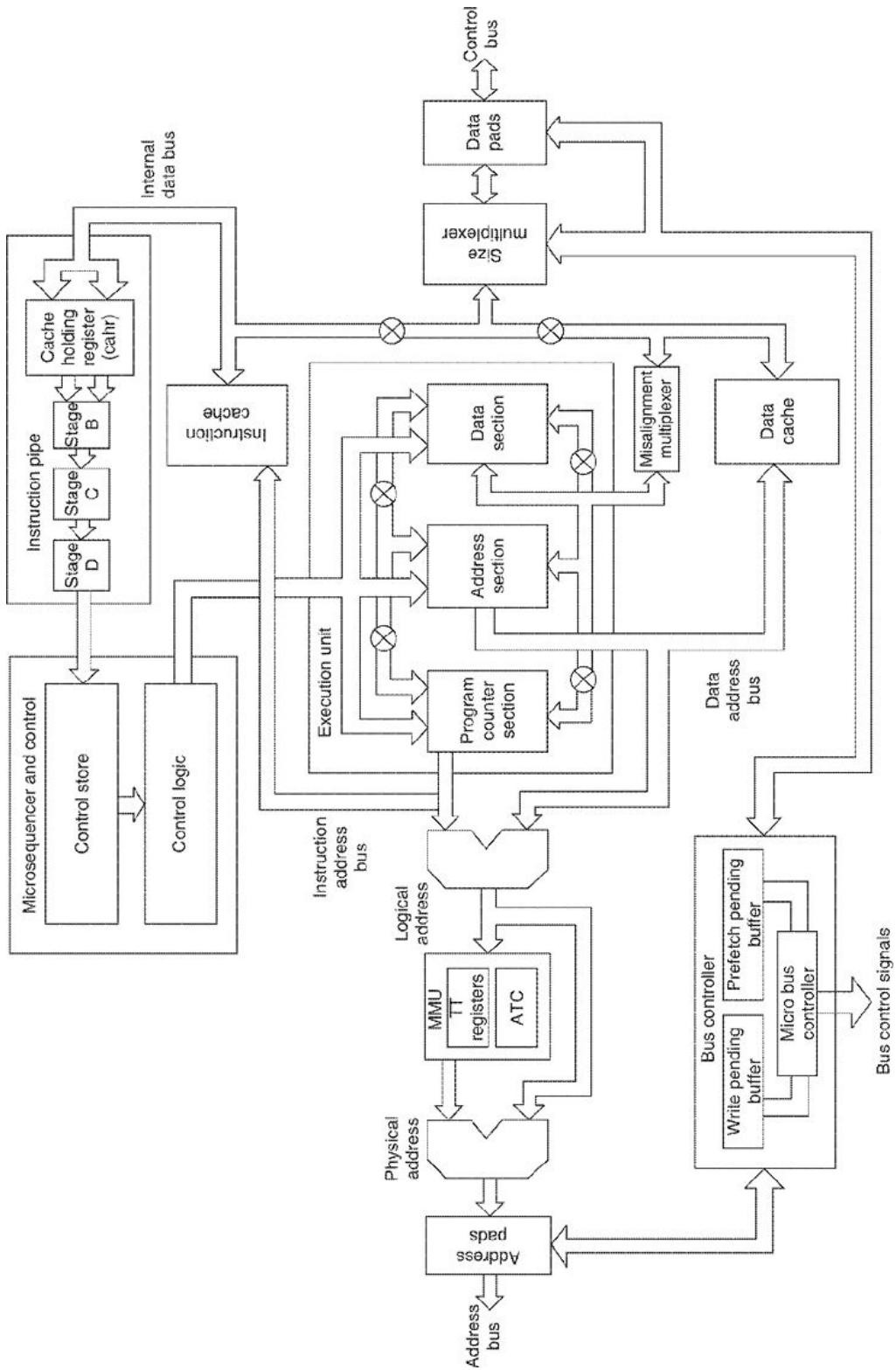


Figure 2.39 Motorola 68030 block diagram.

Intel iAPX 486: iAPX 486 is an advancement over iAPX386 microprocessor series of Intel. It contains pipelined structure having arithmetic logic unit, cache unit, bus interface, instruction decode, prefetch check and floating point unit. It is upward compatible with Intel 8086 and is widely used (Figure 2.40).

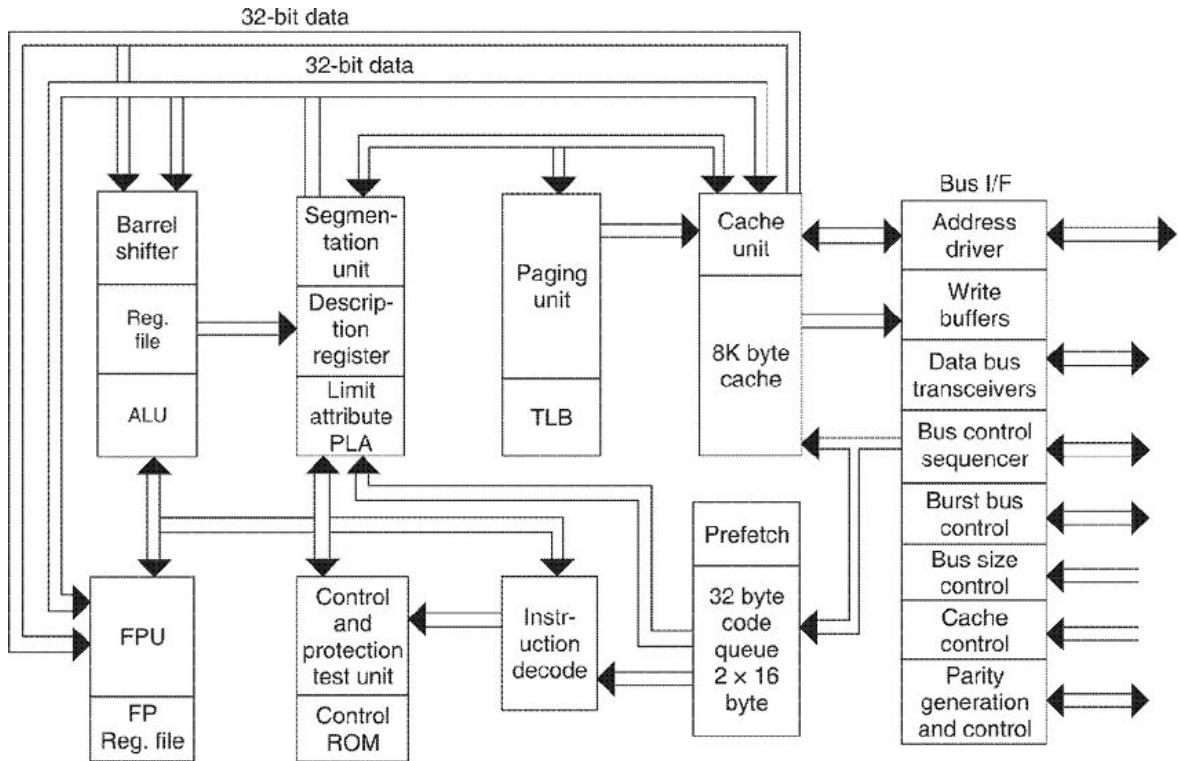


Figure 2.40 iAPX 486 block diagram.

Intel's Pentium processor: The Pentium processor, the newest and the most powerful member of Intel's X86 microprocessor family, incorporates features and improvements made possible by advances in semiconductor technology. A superscalar architecture (Figure 2.41),

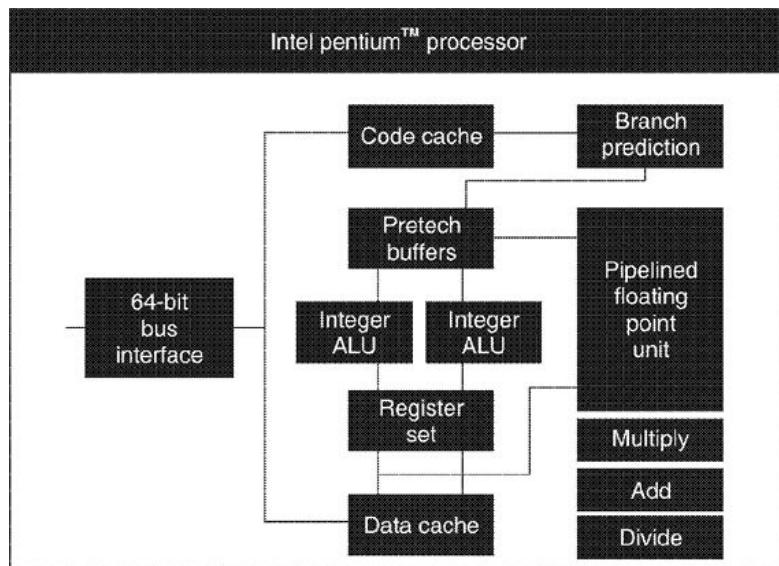


Figure 2.41 Intel Pentium processor architecture.

improved floating point unit, separate on-chip code and write-back data cache, a 64-bit external data bus and other features like branch prediction, multiprocessing support, etc. provide platform for high performance computing.

Intel i860: It is a 64-bit microprocessor that delivers the kind of power and capability associated with supercomputers. It integrates supercomputer features like 64-bit

architecture, parallelism and vector processing and takes full advantage of advanced design techniques like reduced instructions set computing, pipelining, score boarding, by-passing, delayed branching, caching and hard-wired 3-D graphic instructions. It incorporates a risk integer unit, a floating point unit, a 3-D graphic processor, data and instruction cache, memory management and a bus control unit (Figure 2.42).

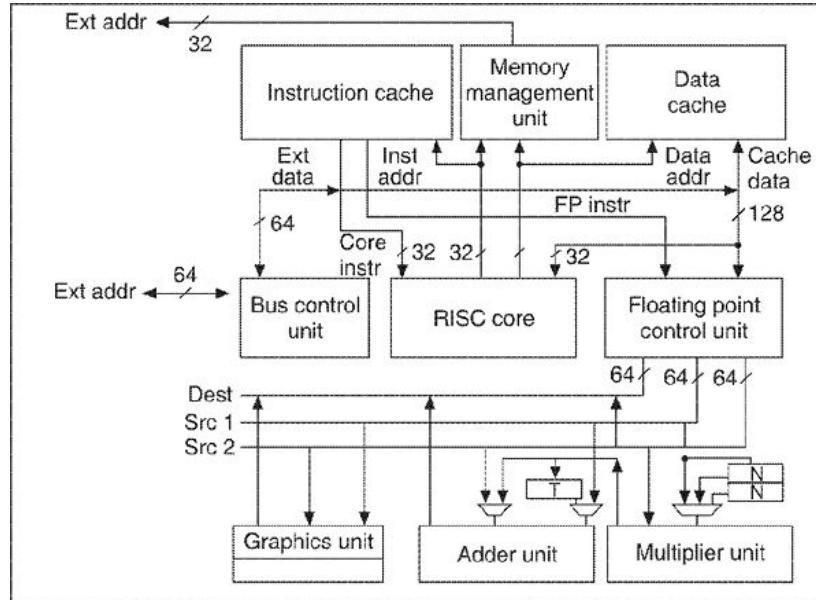


Figure 2.42 Intel i860 block diagram.

2.11.4 Bit-slice Processor

AM 2901 bit-slice processor is available in a slice of 4 bits. Depending on the word size required, more than one slice can be connected to form a higher-bit processor, i.e. an 8-bit processor can be obtained by connecting two slices, a 12-bit processor by connecting three slices, and so on. The IC chip is based on bipolar technology. Thus these processors are faster than processors based on N-MOS technology. User level microprogramming is offered in bit-slice processor. These are most suited for special applications.

2.11.5 Microcomputers and Microcontrollers

A real-time application involves:

- Converting analog signals from transducers to digital value
- Processing of such digital values for limit checking and control algorithms
- Accepting digital information like on-off status of machines directly
- Generation of interrupts to mark an event
- Generation of time signals to time sequence and schedule the events
- Generation of output analog voltage to drive the actuator
- Mechanism for self-checking at runtime (Watch dog timer facility).

Thus a microprocessor dedicated for real-time applications must provide the above facilities. Microcomputers and microcontrollers are specialized microprocessors designed for such applications.

Microcomputers are microprocessors with on-chip memory. Some microcomputer chips contain timer/counter, interrupt handling, along with processor and memory. Timer/counter and interrupts are useful for control applications and these microcomputers are called microcontrollers. In some cases analog-to-digital converter and digital-to-analog converter have also been integrated on the chip.

There is a variety of microcomputers/microcontrollers from different manufacturers like 8048, 8051 and 8096 series of Intel, Z8 from Zilog, M68HC11 from Motorola, 1650 series from General Instruments, IM6100 from Internal Inc. etc. In general, these chips come in three versions, namely “on-chip ROM version”, “on-chip EPROM version” and “ROMless version”. The later two versions are used for development purposes. After the development is complete, mass production of ‘on-chip ROM’ version chips at low cost can be achieved by getting the program fused at source.

Intel 8051 series: Intel 8051 microcontroller (Figure 2.43) is an 8-bit microcontroller with on-chip 8-bit CPU, 4 Kbytes of RAM, 21 special function registers, 32 I/O lines, and two 16-bit timer/counter. It offers 64 Kbytes of address space for external data and 64 Kbytes of address space for external program memory, a five source interrupt structure with two priority levels, a full duplex serial port and bit address capability for Boolean processing. Intel 8031 is a ROM-less Intel 8051 and Intel 8751 is an Intel 8051 with EPROM instead of ROM. Software instructions include powerful multiplication, division, bit set and bit test operations. The hardware, software and application case studies of Intel 8051 have been dealt with in detail in this book.

Intel 8096 series: 16-bit microcontrollers of (MCS-96 series) of Intel (Figure 2.43) are extensions of Intel 8051. The CPU supports bit, byte and word operations including 32-bit double word operation. Four high-speed trigger inputs are provided to record the time at which external events occur, six high-speed pulse units can simultaneously perform timer functions. Up to four such software timers can be in operation at the same time.

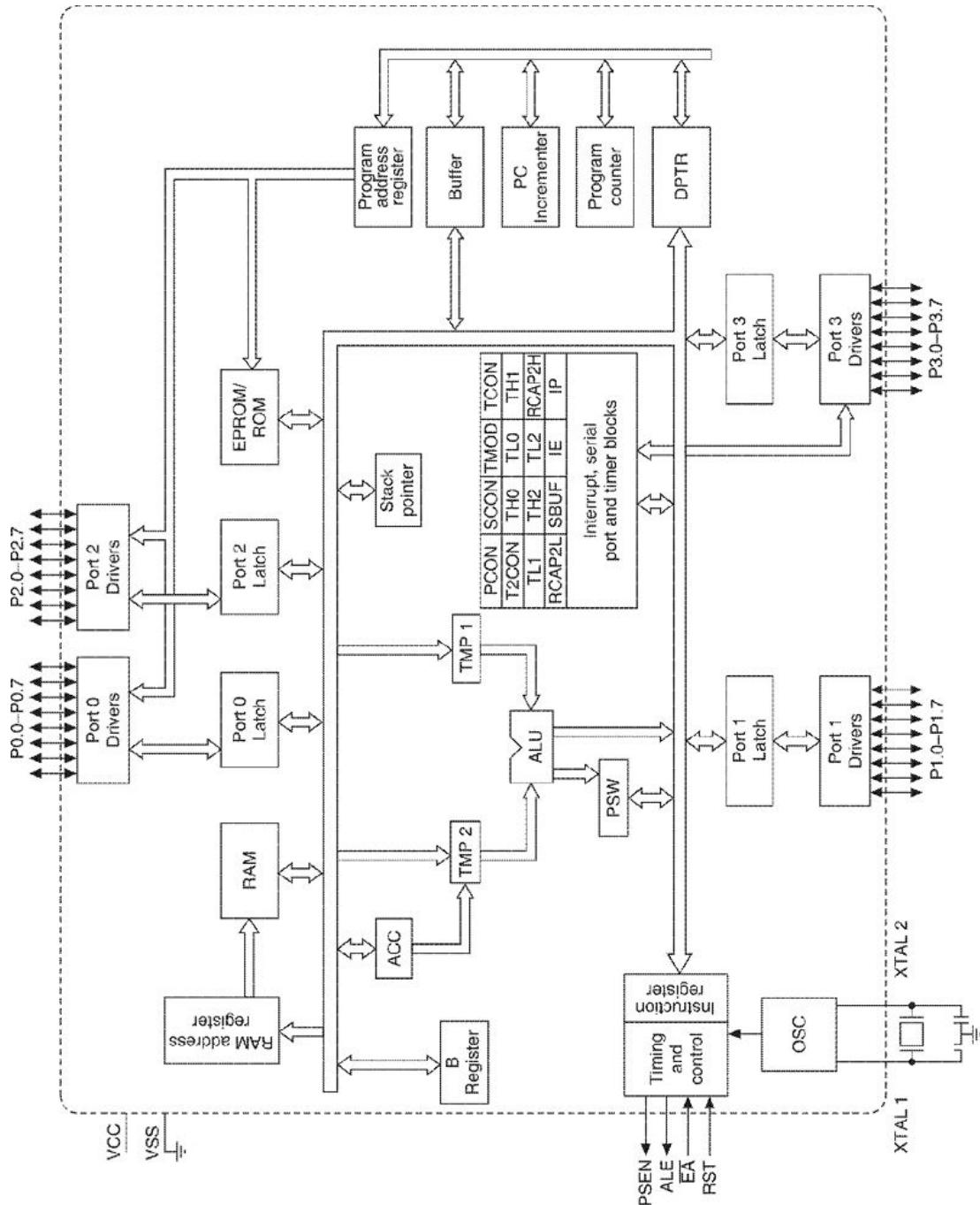


Figure 2.43 Intel 8051 block diagram.

An on-Chip A/D converter converts up to 4 (Intel 8095, Intel 8395) or 8 (Intel 8097, Intel 8397) analog input channels to 10-bit digital value. Also provide on chip are a serial port, a watch dog timer and a pulse width modulated output signal. Eight Kbytes of on-chip RAM is available in case of Intel 8396, Intel 8394, Intel 8397 and Intel 8395 whereas Intel 8096, Intel 8094 and Intel 8095 are ROM-less versions. As stated earlier, A/D conversion is available only with Intel 8095, Intel 8395 Intel 8097 and Intel 8397. The hardware architecture, software and application case studies of Intel 8096 are dealt with in detail in this book.

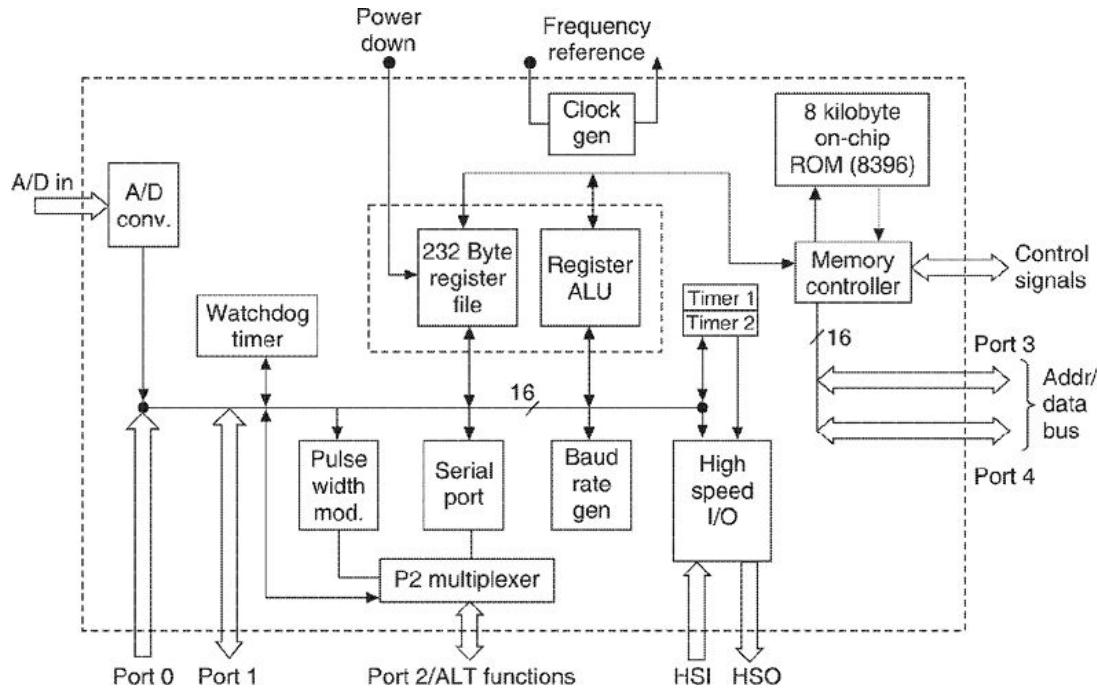


Figure 2.44 Intel 8096 block diagram.

Motorola 68HC11: The Motorola 68HC11 is an 8-bit microcontroller (Figure 2.45). It has internal 16-bit address bus. It has at least 512 bytes of EEPROM and is available in more expensive versions with either 2 Kbytes or 8 Kbytes of EEPROM. The Motorola 68HC11 has five parallel ports. Any line, not serving the specialized alternate as shown in the figure, can serve more general functions. The other facilities include 8 channel, 8-bit ADC, serial port, programmable timer, UART port, etc.

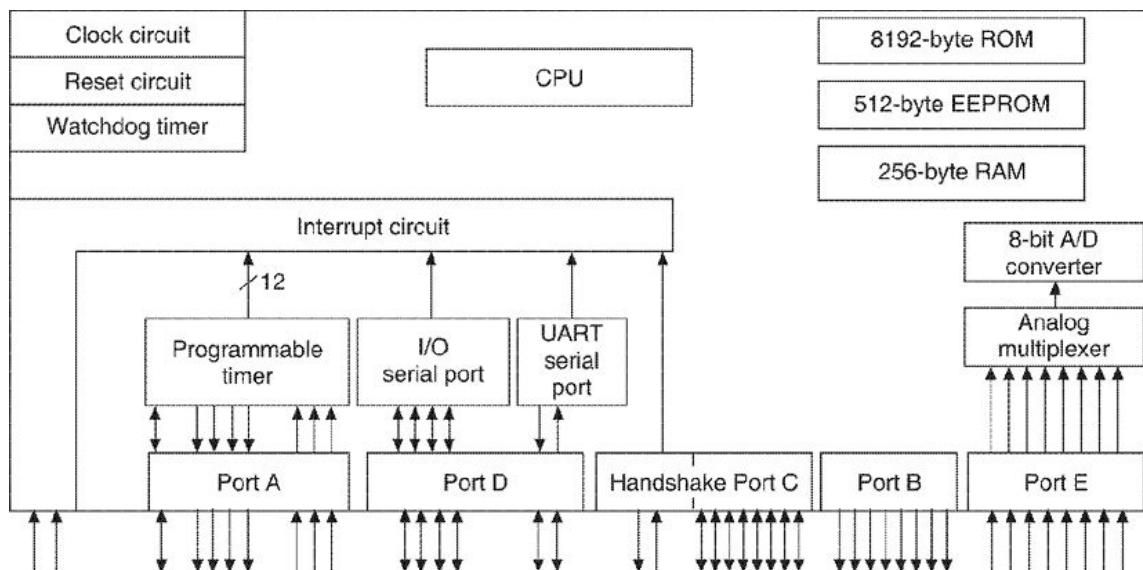


Figure 2.45 Motorola 68HC11 microcontroller architecture.

2.11.6 The Transputer

The principle behind the design of the transputer is to provide the system designer with a building block component which can be used in large numbers to construct very high performance systems. The transputers have been specifically developed for

concurrent processing. The on-chip local memory assists in eliminating processor-to-memory bottlenecks and each transputer supports a number of asynchronous high-speed serial links to other transputer units. The efficient utilization of the processor's time-slices is carried out by a micro-coded scheduler.

The transputer-to-transputer links provide a combined data communications capacity of 5 Mbytes/s and operate concurrently with internal process. This is a radical difference from the shared bus concept employed in the majority of multiprocessor architectures. It allows parallel connection without overhead because of the complex communication between conventional parallel processors. Following are the advantages of transputer over multiprocessor buses.

- No contention for communication.
- No capacitive load penalty as transputers are added.
- The bandwidth does not become saturated as system increases in size.

The system supports high-level concurrent programming languages such as Occam which is specifically designed to run efficiently on transputer systems. The Occam allows access to machine features and removes the need for a low-level assembly language.

Figure 2.46 shows the architecture of IMS "T-800 T-30" transputer. The main features of this chip are (a) integral hardware 64-bit floating-point unit, (b) 2.25 sustained megaflops/s, (c) full 32-bit transputer architecture, (d) 4 Kbytes on-chip RAM for 120 Mbytes/s, (e) 32-bit configurable memory interface, (f) external memory bandwidth of 40 Mbytes/s, (g) high performance graphics support, (h) single 5 MHz clock input-DRAM refresh control, (i) four 10/20 megabit sec, INMOS serial links, (j) external event interrupt—internal timers, and (k) support for runtime error diagnostics, boot from ROM or link, etc.

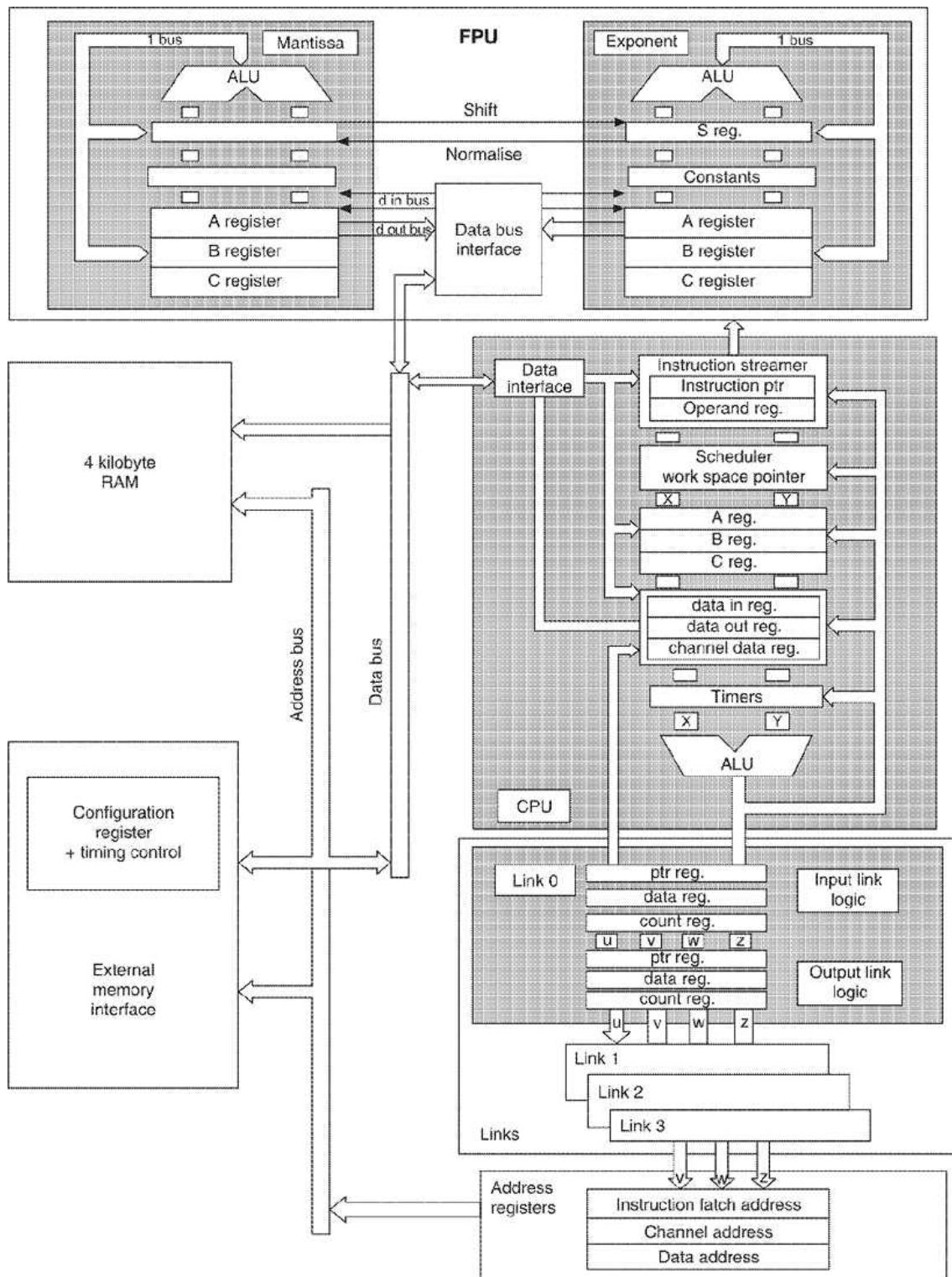


Figure 2.46 T-800 T-30 architecture.

2.12 CONCLUSION

The world of microprocessors is quite large and is changing continuously. In the present chapter, we have dealt with the basic facilities the microprocessor should provide. These facilities dictate the microprocessor architecture and its advancements.

Specialized microprocessor chips for control and digital signal processing are also prevalent in the market apart from generalized chips.

EXERCISES

1. Draw the block diagram of a typical computer organization. Describe the function of each unit. List out the information flow sequence between the different units.
2. What are the basic constituents of Arithmetic Logic Unit (ALU)? Describe the utility of Accumulator and Data Register through examples.
3. What is contained in status register? Explain the utility of status register through an example.
4. What is random access memory? Describe the steps in which data is read from and written into a random access memory. If the number of bits in the address register is 20, what is the size of main memory?
5. Describe the instruction cycle along with the roles of different registers in control unit. For a 3-bit instruction code, draw a block circuit diagram of instruction decode using gates.
6. In a computer:
 - (a) PUSH instruction is used to push the data contained in S register to stack.
 - (b) POP instruction pops the data from stack and stores in R register.
 - (c) The stack is maintained in memory starting from location FFFFH.
 - (d) The present value in stack pointer is FF09H. Values 09, 07, 06, 0A, 01, F0, 0E are stored in memory locations FF09H onwards.
 - (e) The PC points to memory location 1000H. 0E (code for instruction POP) is stored at 1000H, 1001H and 1002H.
Using above as an example, explain the instruction cycle for POP instruction along with operations taking place during execution. What will be the contents of SP, PC, S and R registers when execution of instruction at 1002H is completed?
7. What is microprocessor? How is it different from microcomputer? Describe the information flow between microprocessor, memory and I/O units.
8. What is tristating? Explain the utility of tristating in a microprocessor.
9. The data transfer rate of an I/O device 'A' is considerably less than that of the microprocessor. Which data transfer scheme will you use to transfer data between the microprocessor and the I/O device? Draw a flowchart of data transfer operation.
10. What are the advantages/disadvantages of memory mapped I/O over I/O mapped I/O?
11. What are the functions of Memory Management Unit (MMU) in a virtual memory system operation? Explain step-wise translation of virtual address into physical address by MMU.
12. What is page fault and what action is taken by Memory Management Unit when it occurs?

13. In a computer system the CPU has a 3-stage instruction pipeline
- Read instruction
 - Decode instruction
 - Execute instruction
- Each of the pipeline stage takes 2 ms. Thus, one instruction will take 6 ms to execute in a sequential manner. Take the example of 10 instructions and calculate the time saved in instruction execution using pipelining.
14. What is pipeline flushing? How does this occur? How do the modern systems avoid this condition?
15. Explain how SRAM cache memory increases the throughput of DRAM main memory and thus increases computer speed.
16. How does cache controller keep track of main memory locations mapped in cache memory? What actions are taken if there is a miss?

FURTHER READING

Douglas V Hall, *Microprocessor and Interfacing*, Tata Mcgraw Hill,1991.

Krishna Kant, *Computer Based Industrial Control*, Prentice Hall of India, 1997.

Intel Memory Handbook, Intel Corporation, Santa Clara.

Tannebaum Andrew S, *Structured Computer Organization*, 5th ed., *Prentice-Hall of India, New Delhi, 2005*.

3

INTEL 8085 MICROPROCESSOR HARDWARE ARCHITECTURE

3.1 INTRODUCTION

The 8085 has been one of the popular microprocessors of its time. Due to its unique characteristics this microprocessor is still regarded as a standard by both industry and academics for imparting skill sets on microprocessor basics.

The 8085 removed certain architectural disadvantages of its predecessor, the 8080. The 8085 also provided some additional features over and above the 8080. The 8085 is software compatible to 8080, except for RIM and SIM instructions which are not present in the 8080. The programmable registers are identical in both the processors. The 8085 operates from a single +5 V power supply (as against three power supply lines needed by the 8080); it uses a single clock signal (as against two clock signals used in the 8080) of 320 ns pulse width (500 ns pulse width in case of the 8080). The 8085 has on-chip serial I/O capability as well as interrupt request pins for hardware generated vectored interrupts, which are not present in the 8080.

3.2 HARDWARE ARCHITECTURE

The functional block diagram of the 8085 is shown in Figure 3.1.

3.2.1 The 8085 Clock

An external crystal or *R-C* network can be connected between X₁ and X₂ pins of the 8085 in order to drive the internal clock logic, as shown in Figure 3.2. A clock signal can also be connected directly in a multiprocessor system environment as shown in Figure 3.3.

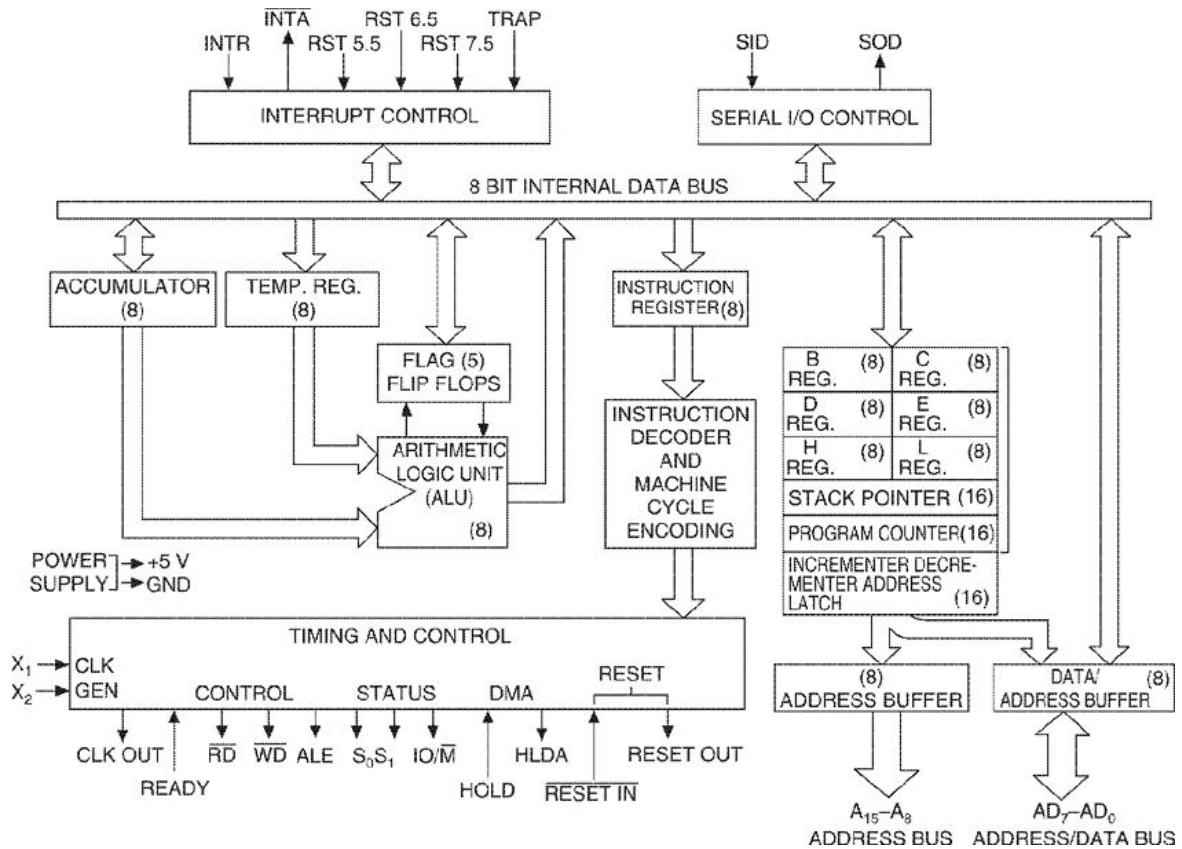


Figure 3.1 Functional block diagram of Intel 8085 microprocessor.

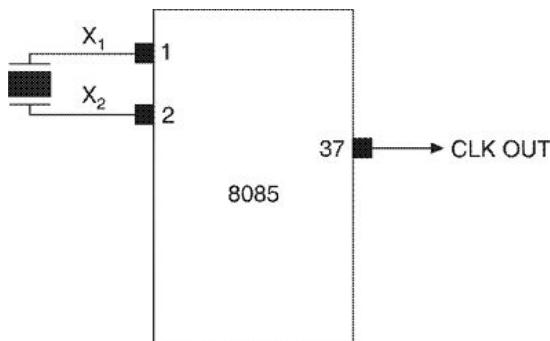


Figure 3.2 Intel 8085 clock connections.

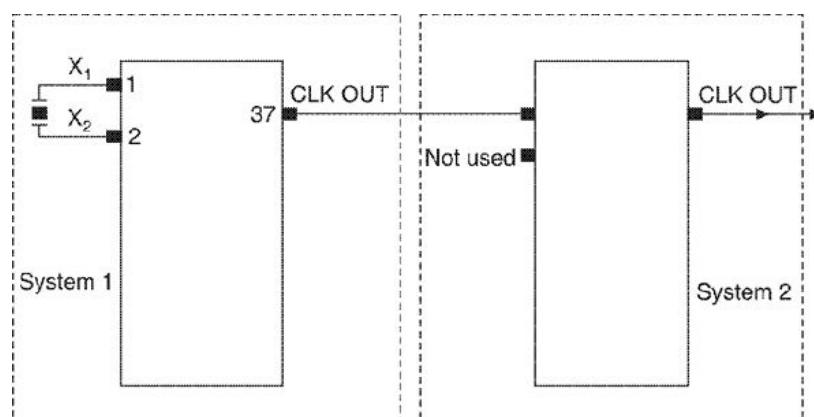


Figure 3.3 Use of CLK OUT in multiprocessor environment.

The input frequency of the clock should be twice the operating frequency. Thus to obtain 320 ns clock (3.125 MHz frequency) the input clock frequency should be 6.250 MHz. A TTL level clock signal is output on the CLK OUT line. This signal is used to drive the slave CPUs in a multiprocessor environment as shown in Figure 3.3, and other devices in the system. The frequency of CLK OUT is the operating frequency, i.e. half of the input frequency. The 8085 comes in several versions. The 8085A expects a main clock frequency of 3 MHz whereas the 8085A-2 expects a main clock frequency of 5 MHz. The 8085AH is the version with 6 MHz frequency and it consumes 20% less power than the 8085A.

3.2.2 Programmable Registers

The general-purpose registers are used to store temporary information during the execution of a program. There is one 8-bit accumulator (abbreviated ACC or A register), and six 8-bit general-purpose registers as shown in Figure 3.4. These registers are named B, C, D, E, H, and L. It is also possible to use these registers in pairs to store 16-bit information. The three pairs which are allowed are BC, DE, and HL.

The 8085 maintains stack in memory. The stack pointer (SP) is a 16-bit register which points to the location of the top of the stack. The stack operation has been explained in Chapter 2. The program counter (PC) is a 16-bit register which points to the next instruction to be executed. The 8085 has 16 address lines. Since both these registers (SP and PC) contain memory addresses, they are 16 bits in length.

In addition to the above registers, there is a 16-bit register called the Program Status Word (PSW). The higher-order 8 bits of PSW contain Accumulator contents and the lower-order 8 bits have the following five condition flags:

- ZERO (Z)
- SIGN (S)
- PARITY (P)
- CARRY (CY)
- AUX. CARRY (AC)

X = Undefined

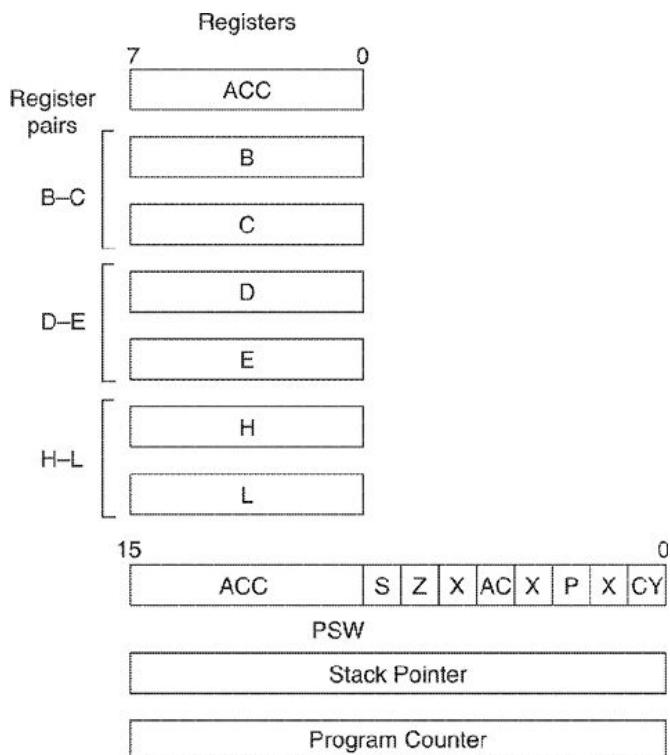


Figure 3.4 Intel 8085 programmable registers.

The flags show the effect of the execution of the last instruction. After the execution of any arithmetic or logic instruction, if the result is zero, the **ZERO** flag is set; if the result is negative, the **SIGN** flag is set; if the result is even parity (number of 1s in the result is even), the **PARITY** flag is set; if there is a carry out of the seventh (MSB) bit, the **CARRY** flag is set; and if there is a carry from the third bit to the fourth bit (i.e. lower nibble to higher nibble), the **AUX. CARRY** flag is set.

A flag is set by forcing the bit to 1 and a flag is reset by forcing the bit to 0. These flags are called condition flags. These condition flags may be used to alter the flow of the program.

3.2.3 Address and Data Buses

The 8085 has 16 address lines and 8 data lines. The data lines are multiplexed with lower address lines. Thus, AD₀ to AD₇ and A₈ to A₁₅ form the 16 address lines of the address bus and AD₀ to AD₇ are bi-directional data lines. These are tristate lines, i.e. they can go to the high impedance state when desired. The tristating has been explained in Chapter 2.

Since the lower-order 8 lines contain the address and data information, both at different times, it becomes necessary to know whether the bit pattern present on lower-order lines pertains to address or

data. The 8085 outputs an Address Latch Enable (ALE) signal which indicates the presence of address on AD₀–AD₇. The falling edge of ALE is used to latch the address present on lower-order address lines AD₀–AD₇ (Figure 3.5).

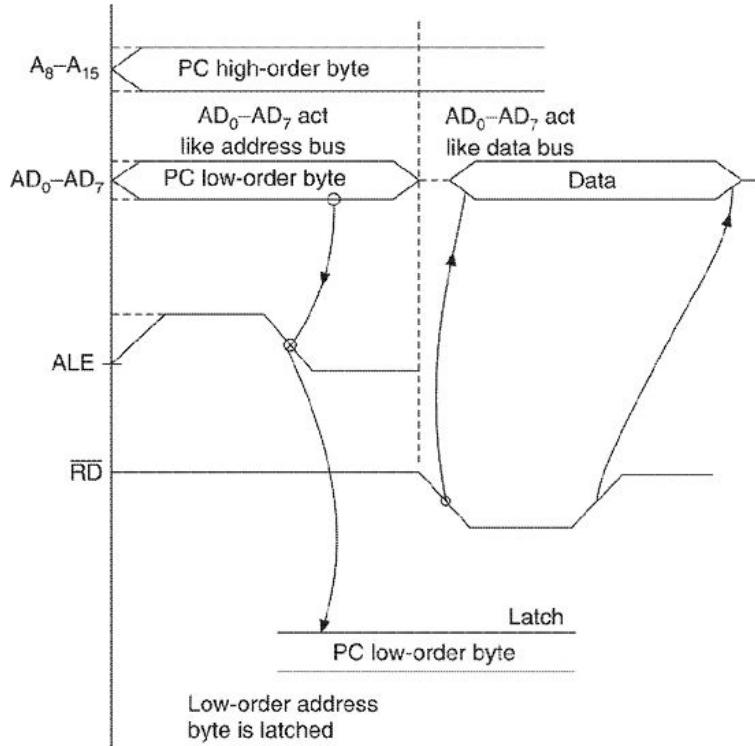


Figure 3.5 Time multiplexing of data bus.

3.2.4 Memory Interfacing

With the available knowledge, let us try to interface the 8085 microprocessor to memory, as it is the main step in the design of a microprocessor-based system. The following logically clear points need to be kept in mind.

- Since the 8085 has 16 address lines, up to 64 KB memory space may be interfaced. A number of memory chips both ROM and RAM can be connected within this space.
- The 8085 will have to issue Read and Write control signals to memory to carry out these operations.
- Since the lower-order address lines AD₀ to AD₇ are used for data as well, in time-multiplexed function, there is the necessity to have a signal that informs us as to whether these address lines contain data or address.
- Since a number of memory chips can be connected and the use of memory space can be planned based on the application, there

would be the necessity of generating a chip-select signal for the memory chip where the Read/Write operation is to be performed. This is done by decoding the address bits and determining the chip where the specified memory address is located.

The block diagram of the address decoder chip 74LS138 which is widely used is shown in Figure 3.6.

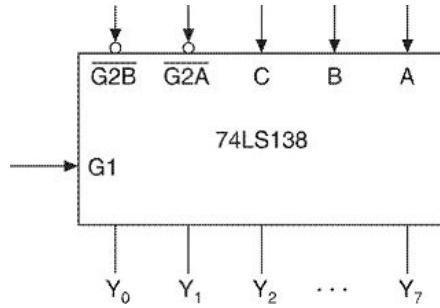


Figure 3.6 Block diagram of 74LS138 decoder.

The decoder is selected when $G1 \cdot \overline{G2B} \cdot \overline{G2A} = 1$.

When selected, the input signals at A, B, C, get translated to Y_0 to Y_7 in the following manner.

C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

But how does it help us? Consider that a 2 KB chip 2716/6116 is to be interfaced to the microprocessor for addresses starting from 0000 to 07FFH. In this case A_{15} , A_{14} , A_{13} , A_{12} , and A_{11} are all equal to 0.

Thus 2716/6116 can be interfaced to 8085 in the manner shown in Figure 3.7.

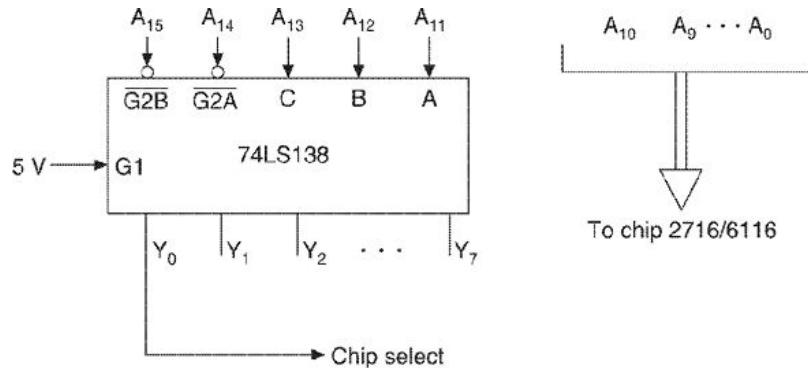


Figure 3.7 Generation of chip select signal for memory.

Now suppose that the address space for 2716/6116 starts at 1000H. In this case, since A₁₂ = 1, Y₂ = 1 can be used as the chip select signal. Figure 3.8 shows the interfacing of 2716/6116 and 2732 memory chips to the 8085.

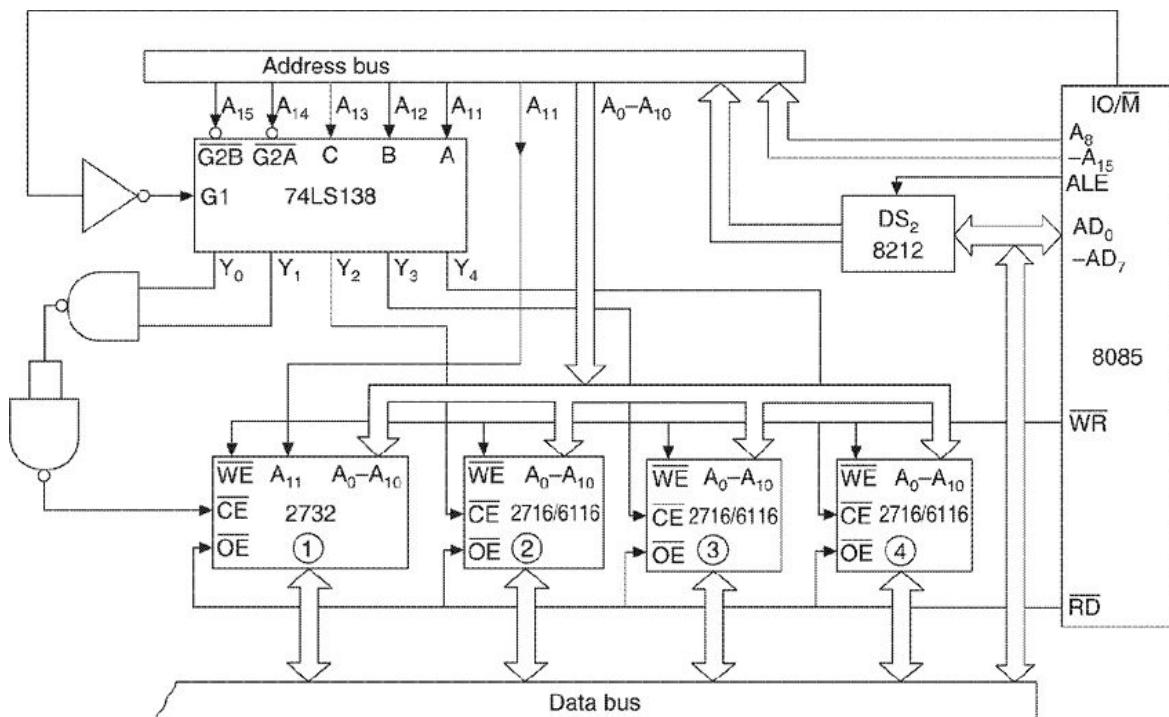


Figure 3.8 Memory interfacing with Intel 8085.

The 8212 has been used to latch the lower-order address on address bus through the ALE signal. The total memory space is 10 KB. The address lines A₀ to A₁₀ are connected to memory chips. The lines A₁₁ to A₁₅ are decoded by 74LS138 decoder to generate the chip select signals for various memory chips. 74LS138 is selected when $G1 \cdot (G2A \cdot G2B) = 1$, i.e. when IO/\bar{M} signal is low, i.e. when memory operations are to be performed by the processor. The \bar{RD} is connected to \bar{OE} of each memory chip and \bar{WR} to \bar{WE} of each chip. The address range

for various chips can be decoded very easily by the status of signals A15, A14, A13, A12 and A11 of the address bus, as given below.

Memory chip	Address range	Address lines status				
		A15	A14	A13	A12	A11
Chip 2732 (1)	0000–0FFF (4KB)	0	0	0	0	1
Chip 2716 (2)	1000–17FF (2KB)	0	0	0	1	0
Chip 2716 (3)	1800–1FFF (2KB)	0	0	0	1	1
Chip 2716 (4)	2000–27FF (2KB)	0	0	1	0	0

There are three signal in this group, namely \overline{RD} for read operation, \overline{WR} for write operation and $\overline{IO/M}$ to indicate whether it is memory read-write or IO read-write. Clearly, the \overline{RD} signal will be pulsed Low for read operation after the address information has been transferred to address lines. Simultaneously, $\overline{IO/M}$ will be High or Low (High for I/O and Low for memory) depending on memory or I/O read. After this, the data lines AD₀–AD₇ will contain the data.

Similarly, after the address has been transferred to AD₀–AD₇, A₈–A₁₅ and subsequently the data has been transferred to AD₀–AD₇, the \overline{WR} is pulsed Low to perform the write operation. Simultaneously, $\overline{IO/M}$ assumes High or Low state depending on whether the data is to be written in memory or in I/O device.

Often the memory chips or the I/O device are slow in nature. These devices take more time compared to 8085 for read/write operations. Thus after the \overline{RD} control signal has been issued by 8085, there will be some time-gap before the data will appear on data lines. So is the case with the write operation.

Slow memory or I/O devices can gain additional time of the 8085 by inputting the READY signal. This input can be used to insert **wait state** clock periods in any machine cycle. Effectively, the 8085 waits till this signal is withdrawn and then performs the normal operation. This signal is also referred as **Wait State Request** signal.

3.2.5 Interrupt System

Six signals are associated with interrupt logic. There are five interrupt request pins through which the 8085 may be interrupted. These are TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR. The first four interrupts (TRAP, RST 7.5, RST 6.5, RST 5.5) are supported by hardware implemented vectoring, i.e. the locations of Interrupt Service Routines

(ISR) in memory for these interrupts are fixed and automatic vectoring to the location takes place whenever any of these four interrupts is recognized. The interrupt service routine locations for these interrupts are as follows:

Interrupt	ISR location
TRAP	24H
RST 7.5	3CH
RST 6.5	34H
RST 5.5	2CH

The TRAP is a non-maskable interrupt, i.e. it cannot be disabled while all other interrupts can be disabled or enabled selectively or collectively.

The last interrupt INTR is a general-purpose interrupt which is available on the 8080 as well. The interrupt request on this line is acknowledged by the 8085 on the INTA output line. On receiving the acknowledgement, the interrupting device places on the data bus the address of Interrupt Service Routine (ISR) in the memory. The microprocessor executes the ISR to service the interrupt. There are two ways by which the interrupting device can send the ISR address. It can input one byte code of one of the “RST n” instructions (RST 0 to RST 7). The RST n is a single byte subroutine call instruction. The microprocessor will execute this instruction to branch to ISR at $8 \square n$ address location (refer to RST instruction in the next chapter). Another way is to input a 3-byte code for “CALL addr16” instruction. On receiving this 3-byte code, the microprocessor will execute this instruction to branch to ISR at addr16.

Interrupts INTR, RST 5.5 and RST 6.5 are level-triggered, i.e. a high signal should be present on these lines when the interrupt is recognized. In other words, if a device sends an interrupt pulse of short duration on any of these lines, it is possible that this interrupt request is ignored totally owing to the reason that the interrupt pulse may not be present when these lines are scanned by the microprocessor for the interrupt request.

RST 7.5 is edge-triggered and TRAP is level- as well as edge-triggered. Both are associated with internal latches which store the interrupt requests as and when the interrupt signal on these lines goes from low to high. TRAP is normally reserved for power failure or some other highest priority interrupt. Thus, it will be desirable to protect this pin from spurious signals. It is because of this reason that TRAP is level-

as well as edge-triggered. The priority structure of interrupts is as follows:

TRAP
RST 7.5
RST 6.5
RST 5.5
INTR

It means that when more than one interrupt is pending and enabled, the higher priority interrupt will be recognized (if enabled).

Enabling, disabling and masking of interrupts

There are two software instructions EI and DI associated with the interrupt system. The EI instruction enables the interrupts while the DI instruction disables all the interrupts except TRAP. Once any interrupt has been acknowledged, the interrupt system is disabled automatically. However TRAP is not disabled even now. Because of this, TRAP is called the highest priority interrupt. A high level on INTR line can be recognized only if:

- (a) The TRAP and RST lines are not high, i.e. no interrupt is pending on these lines.
- (b) The interrupt system, if disabled previously, is enabled by EI instruction.

Interrupt masking

Various interrupts can be enabled or disabled selectively by masking. There is an interrupt mask register in the interrupt control system (Figure 3.9). Various bits of this register are used for enabling or disabling of interrupts. The first 3 bits—bit 0, bit 1, and bit 2—are devoted to interrupt masks for RST 5.5, RST 6.5 and RST 7.5 interrupts respectively. If any of these bits is set to 1, the particular interrupt is disabled, otherwise it is enabled. Bit 3 is the overriding bit for bit 0, bit 1, and bit 2. This bit is called the Mask Set Enable (MSE). Only when this bit is set to 1, the mask bits for interrupts RST 5.5, RST 6.5, and RST 7.5 are taken into consideration. Bit 4 is used to reset the RST 7.5 latch. When this bit is set to 1, the latch associated

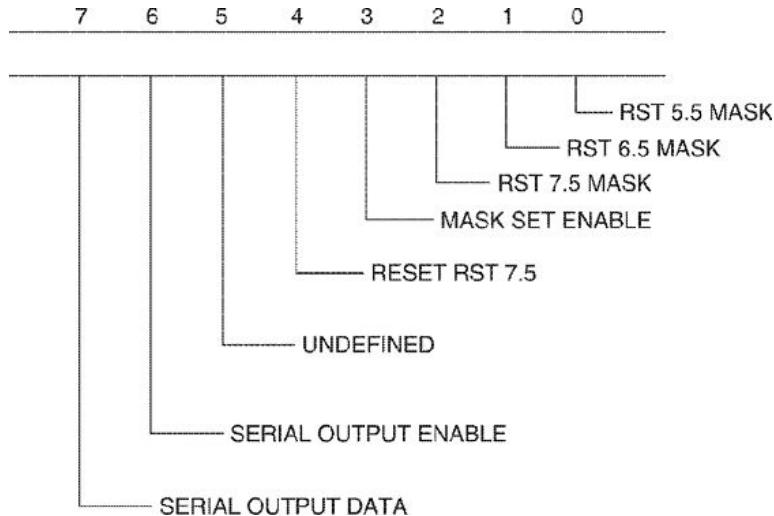


Figure 3.9 Interrupt mask register.

with the RST 7.5 interrupt is reset regardless of whether RST 7.5 is masked or not. Bit 5 is not used. Bit 6 and bit 7 are devoted to serial output; these bits are discussed under Section 3.2.7 on Serial Input–Output. All the interrupts can also be reset/disabled through the signal at the RESETIN pin.

To mask any interrupt, the user will set the bit pattern in accumulator (ACC) according to his need and execute the SIM (Set Interrupt Mask) instruction. As an example, if we want to enable RST 7.5 and RST 5.5 and disable RST 6.5, the bit pattern will be

$$0\ 0\ 0\ 1\ 1\ 0\ 1\ 0 = 1AH$$

The user will transfer 1AH to ACC and execute the SIM instruction.

The user will often wish to know about the status of the various interrupts, i.e. whether they are enabled and whether any interrupt is pending on these lines. This is facilitated by the RIM (Read Interrupt Mask) instruction. This instruction when executed, loads the accumulator with a byte—the structure of which is shown in Figure 3.10. Bit 0, bit 1, and bit 2 of this byte indicate interrupt masks set for interrupts RST 5.5, RST 6.5, and RST 7.5 respectively. Bit 3 shows whether the interrupt system is enabled or disabled. The next three bits indicate whether any interrupt is pending on RST 5.5, RST 6.5, and RST 7.5 respectively. Bit 7 is reserved for serial input data.

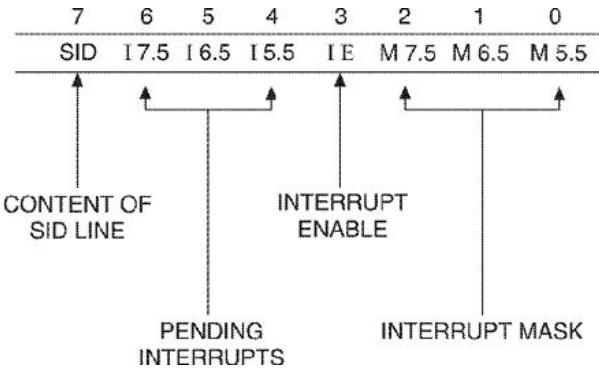


Figure 3.10 Interrupt pending register.

The user will examine the status of various interrupts and then take appropriate decision to mask or unmask any interrupt.

3.2.6 Direct Memory Access

Two signals are associated with Direct Memory Access—HOLD and HLDA. The HOLD is used for DMA request. It indicates that another master is requesting for the use of address and data buses and control signals ALE, RD, WR, IO/M. On receiving the HOLD signal, the CPU completes the current machine cycle and then relinquishes the use of buses. The CPU sends HLDA to acknowledge to device that HOLD request has been received and it will relinquish the buses in the next clock cycle. When HLDA goes high, the address, the data, RD, WR, ALE and IO/M lines are tristated. The internal processing in CPU may however continue. The processor can regain the buses only after the HOLD is removed. HLDA goes low after the HOLD is removed. The CPU takes over the buses—a half clock-cycle after HLDA goes low.

3.2.7 Serial Input–Output

Serial input–output is facilitated via two lines. These lines are Serial Input Data (SID) and Serial Output Data (SOD). As the name implies, the SID line is used by the serial output devices to send bit serial data to processor and SOD is used by the processor to output bit serial data to serial input devices. There are two software instructions which are associated with SID and SOD lines.

The RIM (Read Interrupt Mask) instruction transfers the bit information present on the SID line to the seventh bit of ACC (Figure 3.10). The programmer takes out this bit for further processing. The SIM (Set Interrupt Mask) instruction transfers the seventh bit of ACC to SOD line (Figure 3.9). The serial input device accepts this bit information. It

remains the responsibility of the programmer to fix the bit in the seventh bit of ACC before the SIM instruction is executed.

3.2.8 The 8085 Activity Status Information

Two signals S_0 and S_1 define the activity on the buses as follows.

S_1	S_0	<i>Operation specified</i>
0	0	Halt
0	1	Memory or I/O Write
1	0	Memory or I/O Read
1	1	Instruction Fetch

When coupled with the IO/M signal, it is possible to define the exact operation. These signals are useful in debugging a circuit in which the 8085 is one component. In addition, S_1 may also be used as an advance read-signal, since it precedes the actual read ($\overline{\text{RD}}$) control signal.

3.2.9 The 8085 Reset

Two signals are associated with the reset logic. $\overline{\text{RESETIN}}$ input signal, when goes low, resets the processor. The program counter is set to zero, and interrupt enable and HLDA flip-flops are reset. The program execution starts at zero location. The $\overline{\text{RESETIN}}$ input signal need not be synchronized with the clock.

The CPU outputs the RESET OUT signal which is synchronized with the clock. It may be used to reset the other associated circuits.

3.3 THE 8085 PIN OUT

The 8085 microprocessor is available on a 40-pin Dual-in-Line Package (DIP). The pin configuration is shown in Figure 3.11.

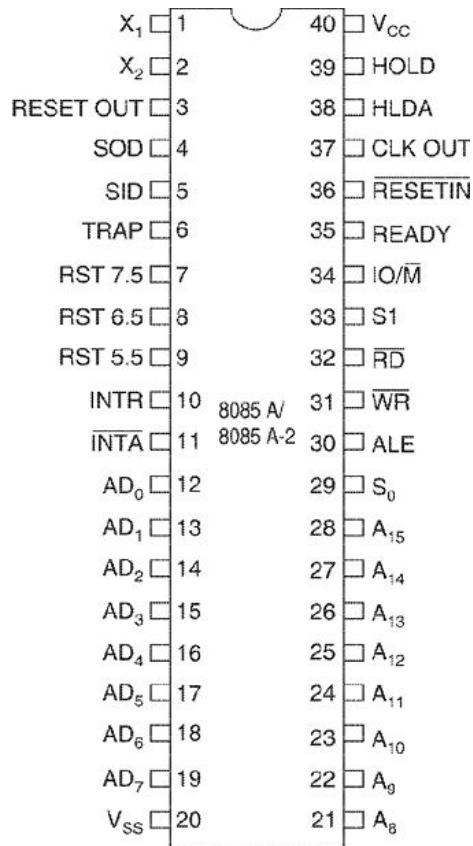


Figure 3.11 Intel 8085 pin diagram.

3.3.1 The 8085 Signals

Even though we have dealt with all the signals while describing the functions of the 8085, the description of Pins and Signals is summarized here for ease of reference.

Pin no.	Name	Type	Functional description
1, 2	X ₁ , X ₂	Input	A clock oscillator crystal is connected between these two pins to provide a timing signal to the processor. Maximum permissible clock frequency is 6 MHz (Intel 8085 AH).
37	CLK OUT	Output	Clock signal for other devices connected to the microprocessor is provided on this pin. The frequency of CLK OUT is the operating frequency, i.e. half the input frequency.
12 to 19	AD ₀ to AD ₇	Bidirectional Tristate	These pins provide multiplexed address/data output of A ₀ to A ₇ and D ₀ to D ₇ .
21 to 28	A ₈ to A ₁₅	Output Tristate	High-order address bits A ₈ to A ₁₅ are provided on these pins.
30	ALE	Output Tristate	Address Latch Enable signal is provided on this pin. The falling edge of ALE may be used to latch address from lower-order address lines AD ₀ to AD ₇ .
32		Output Tristate	\overline{RD} (Read control) signal is provided on this pin by the microprocessor for memory or I/O read. Read operation is enabled when \overline{RD} = Low.
31		Output Tristate	\overline{WR} (Write control) signal is provided on this pin by the microprocessor for memory or I/O write. When \overline{WR} = Low, write operation is enabled.

34		Output Tristate	The signal on this pin indicates whether the address on the address bus is for memory or I/O (High = I/O, Low = memory) read/write operation.
35	READY	Input	READY is the signal provided by memory or I/O device to the microprocessor. When READY = High, the microprocessor proceeds to process the data in usual manner. When READY = Low, the microprocessor goes into wait state and waits for READY to become high.
6	TRAP	Input	
7	RST 7.5	Input	
8	RST 6.5	Input	
9	RST 5.5	Input	These pins provide the interrupt signals to microprocessor, with hardware generated vectoring to Interrupt Service Routines.
10	INTR	Input	This pin provides the interrupt signal to the microprocessor. The interrupting device is required to provide the location of <u>Interrupt Service Routine</u> on the data bus, on receipt of the acknowledge signal on the <u>INTA</u> line.
11		Output	When the microprocessor receives an interrupt request on INTR, it stores the contents of the register and then sends a low signal on <u>INTA</u> as interrupt acknowledgement to the interrupting device.
39	HOLD	Input	
38	HLDA	Output	These pins are associated with Direct Memory Access. When a high signal is received at HOLD, the microprocessor suspends the execution of further instructions and tristates the Address bus, Data bus and control signals <u>RD</u> , <u>WR</u> , <u>ALE</u> , <u>IOM</u> . It then issues a high signal at HLDA (HOLD ACKNOWLEDGE). The I/O device can now use the buses and control signals for transferring data to or from memory. The processor regains the buses only when the HOLD signal is removed.
5	SID	Input	
4	SOD	Output	The SID (Serial Input Data) line is used by the serial output device to input serial data, and the SOD (Serial Output Data) line is used by the microprocessor to output bit serial data to serial input device.
29	S ₀	Output	
33	S ₁	Output	The microprocessor outputs signals on S ₀ and S ₁ pins to indicate that the bus activity is in progress.
36	RESET OUT	Input	
3		Output	A low signal at pin 36 (<u>RESETIN</u>) resets the processor. The microprocessor outputs high at pin 36 (RESET OUT) which is synchronized with the clock.
20	V _{SS}	—	This pin provides the ground point for the chip.
40	V _{CC}	—	This pin is used to provide +5 V dc supply required for microprocessor operations.

3.4 INSTRUCTION EXECUTION

As already explained in Chapter 2, the execution of an instruction consists of two distinct parts, namely the Opcode Fetch and the Opcode Execute. Also, all CPU operations are synchronized with the clock. With this background, we shall explain the instruction execution in the 8085.

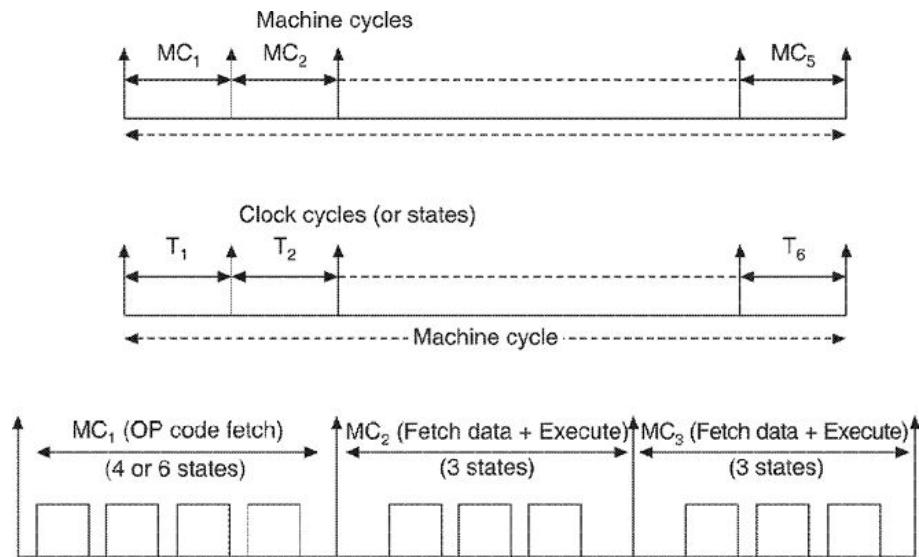


Figure 3.12 Instruction cycle details.

In the 8085, an instruction is executed in 1 to 5 machine cycles (MC₁ to MC₅). Thus an instruction cycle consists of a number of (maximum five) machine cycles. The first machine cycle MC₁ will be the opcode fetch cycle as explained previously. The subsequent machine cycles are used to fetch data and execute the instruction. Each machine cycle consists of a number of clock cycles. These clock cycles (T₁, T₂, T₃, ...) are also called **states**. The first machine cycle (opcode fetch) will have four or six clock periods while the subsequent machine cycles will have only three clock periods (Figure 3.12).

Event sequence timing diagram

To understand the operation of the 8085 during each machine cycle, we will use the event sequence timing diagram which attempts to illustrate event sequences through signal transitions and levels.

Figure 3.13(a) shows the multiple signal level transition. Two parallel lines indicate a group of signals like address or data bus. Some of the signals in this group may change state. The causes and consequences are shown in Figures 3.13(b), (c), and (d). In Figure 3.13(b), a high-to-low transition (cause) of signal A causes low-to-high transition (consequence) of signal B. Figures 3.13(c) and (d) show the multiple causes to a single consequence and a single cause to multiple consequences, respectively.

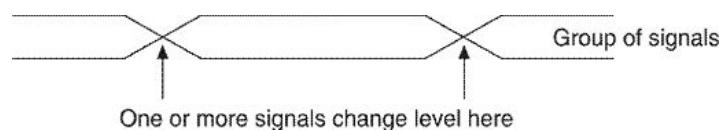


Figure 3.13(a) Multiple signal level transition.

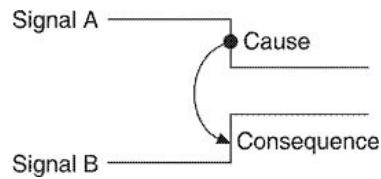


Figure 3.13(b) Single cause to single consequence.

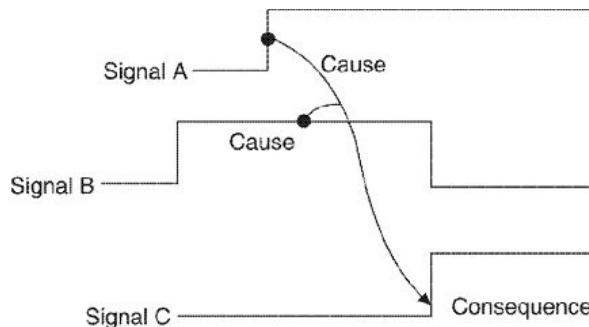


Figure 3.13(c) Multiple causes to single consequence.

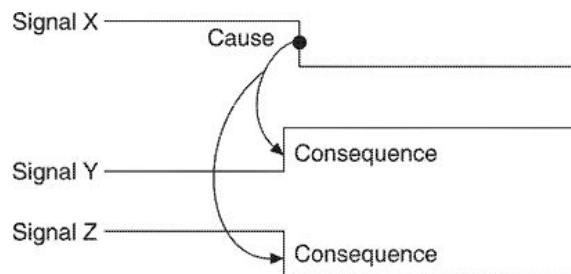


Figure 3.13(d) Single cause to multiple consequences.

Opcode fetch machine cycle

Figure 3.14 shows the timing diagram of the opcode fetch machine cycle, which is the first machine cycle in any instruction execution. During this machine cycle, an instruction is fetched and decoded.

Opcode fetch will involve reading an instruction from memory. Thus $\text{IO}/\bar{\text{M}}$ signal will be low, signifying the memory operation. Both S_0 and S_1 will be high to signify the opcode fetch.

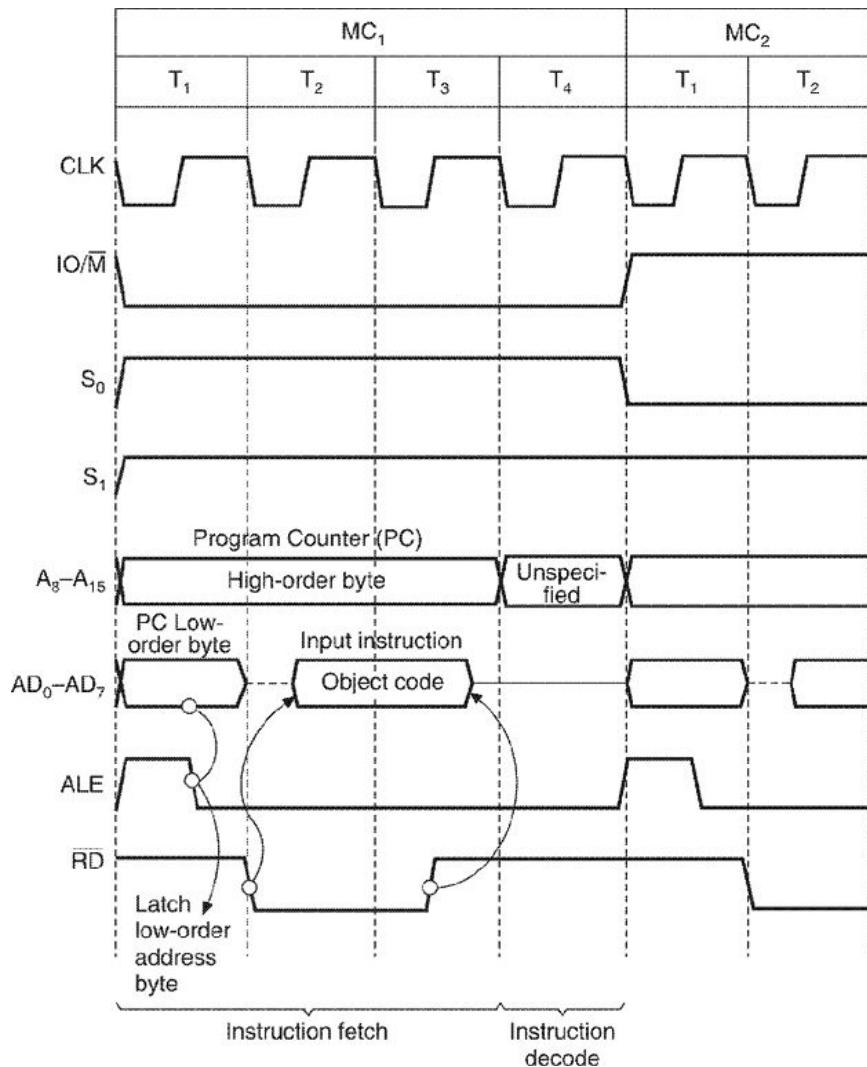


Figure 3.14 A four clock period instruction fetch machine cycle.

The following events will take place during each clock cycle (state).

T₁

- Program counter (having the address of the instruction to be executed) will be loaded to address bus AD₀-AD₇, A₈-A₁₅.
- ALE will be pulsed high to indicate the presence of address on AD₀-AD₇.
- Falling edge of ALE is used to latch the low-order address byte from AD₀-AD₇.

T₂

- \overline{RD} (Read Control signal) goes low for read operation. Combined with $\overline{IO/M}$ which is low for memory operation, it is taken as memory read.
- Data from memory is loaded to data bus. This data is nothing but the object code of the instruction to be executed.

T₃

- \overline{RD} goes high to indicate the end of the read operation.
- Object code is loaded to instruction register in processor.

T₄

- Opcode is decoded.

Through the opcode decode, the microprocessor determines whether its execution will require any memory or I/O access. If yes, then memory or I/O read/write cycles are entered. Otherwise, the instruction is executed in this machine cycle. If required, T₅ and T₆ clock cycles are entered.

Let us consider an example of MOV r₁, r₂. This instruction, when executed, transfers the content of register r₂ to register r₁. (Registers r₁, r₂ may be any of the seven registers A, B, C, D, E, H, and L.) This instruction will require only the opcode fetch cycle, since both the operands are in CPU registers. Data transfer is performed during the T₄ clock cycle of the machine cycle. Thus, this instruction is executed in one machine cycle with four states.

Memory read machine cycle

S₀, S₁ and $\overline{IO/M}$ signals will assume the following status:

S₀ = low and S₁ = high to indicate the read machine cycle,
 $\overline{IO/M}$ = low to indicate the memory operation.

The operation during the first two clock cycles T₁ and T₂ is the same as that of the opcode fetch machine cycle. However in this case, in the T₁ clock cycle, the memory address of the data byte to be read is loaded to the address bus, AD₀-AD₇, A₈-A₁₅.

T₃

- \overline{RD} goes high to indicate the end of the read operation.
- Data is transferred to the register mentioned.

If memory is slow, i.e. if the data cannot be accessed in one clock cycle, it will pull the READY pin low indicating that memory is not ready to transfer the data. The READY pin is sampled in the T₂ clock cycle of each machine cycle. If READY is low then wait states are entered between T₂ and T₃ clock cycles till the READY pin become high (Figure 3.15).

As an example, the “MVI r, Data8” instruction, which moves the data byte stored (along with the instruction) in memory to a register r, will be executed in two machine cycles. The first cycle will be the opcode fetch and the other the memory read machine cycle. Thus the instruction will take seven clock cycles (states) to execute.

In I/O read machine cycle, all the operations are same except that the $\overline{IO/M}$ signal is kept high to indicate I/O operation. The write machine cycles for I/O and memory are similar and can be evolved by the reader.

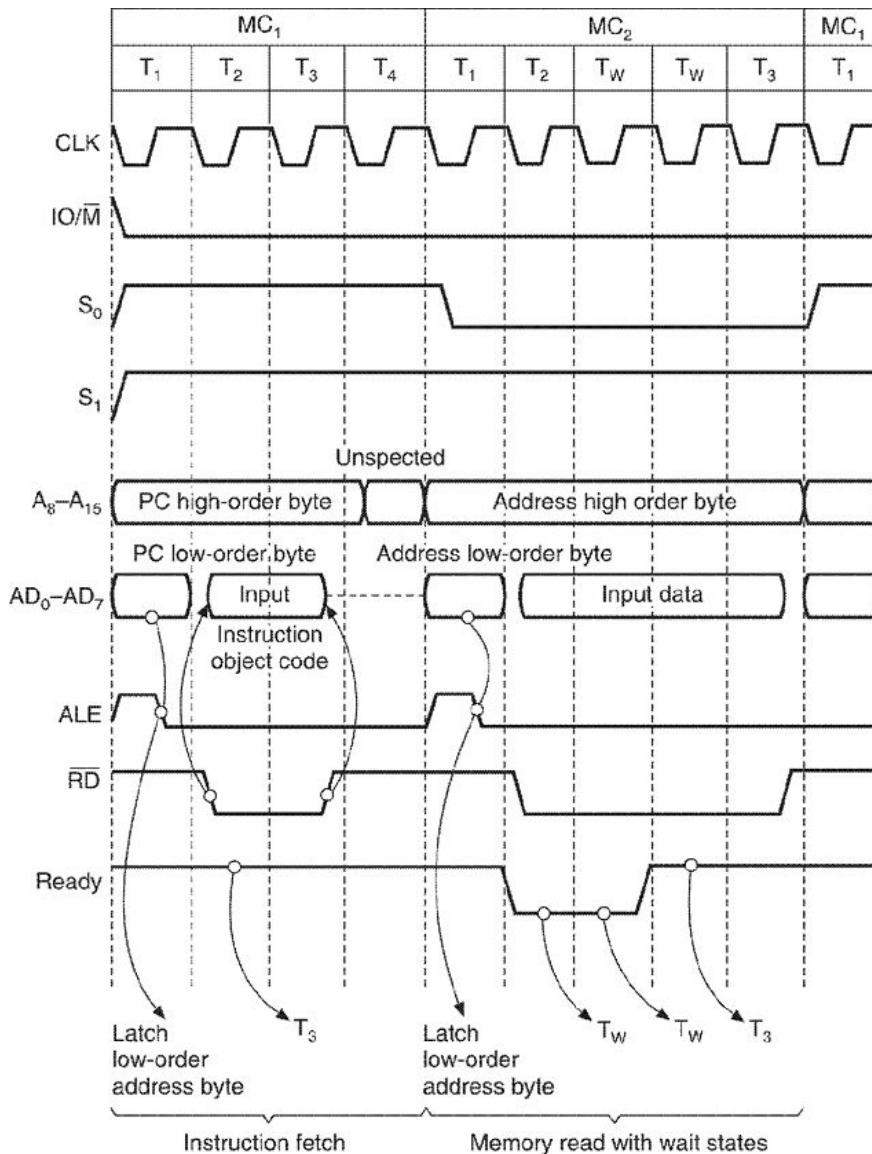


Figure 3.15 Wait state occurring in a memory read machine cycle.

3.5 DIRECT MEMORY ACCESS TIMING DIAGRAM

The HOLD and HLDA signals are used for direct memory access. As explained in Chapter 2, the I/O devices whose speed match with memory, can directly access the memory using the DMA facility. Using DMA, bulk data transfer can take place between the memory and the I/O device, bypassing the microprocessor. The sequence of the operation has also been briefly described in Chapter 2. Here, we shall describe the DMA operation in the 8085A microprocessor.

The 8085A microprocessor makes use of HOLD state for transiently floating the system bus. During HOLD, the external logic gains control of the bus to perform a direct memory access operation. DMA is initiated

by external logic (of the I/O device). It requests for a HOLD state by inputting the HOLD signal high. The microprocessor responds by entering the HOLD state and outputting the HLDA signal high. During a HOLD state, the microprocessor floats all tristate signals.

Figure 3.16 shows the timing diagram of a HOLD state.

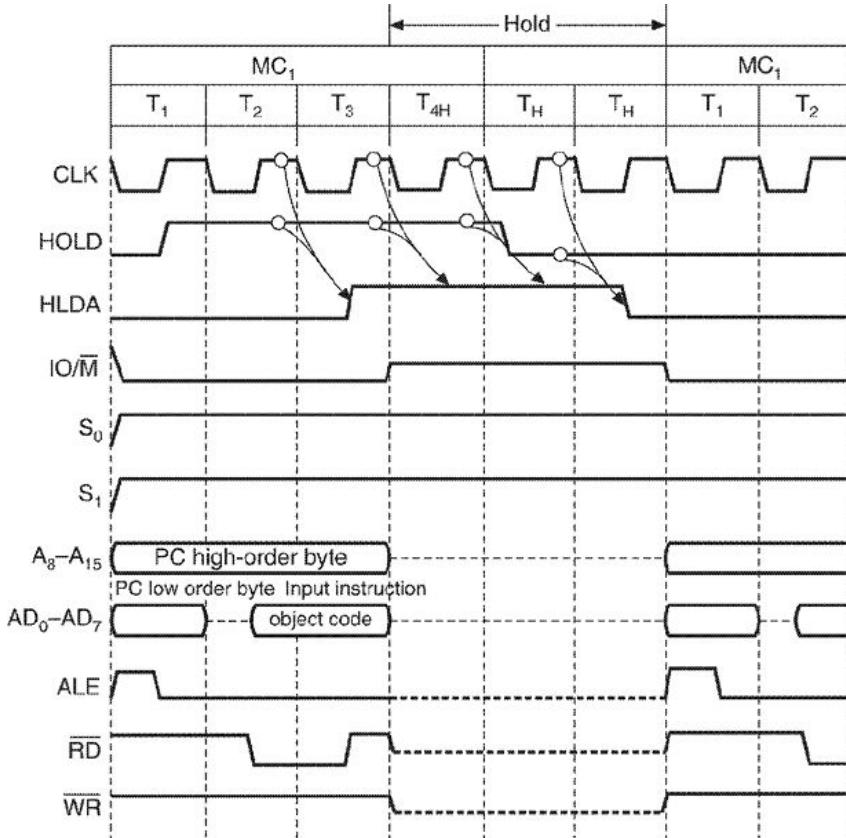


Figure 3.16 DMA operation (HOLD state) following a single-byte instruction.

- The signal at the HOLD pin is sampled during T_2 in each machine cycle.
- If HOLD is high at this time, HLDA (HOLD ACKNOWLEDGE) is output high during T_3 .
- As soon as high level is detected at HOLD (at T_2), a two-clock period HOLD state initiation sequence begins. HOLD state begins at T_{4H} .
- There is no restriction in 8085A on the duration of HOLD state which lasts as long as the HOLD input is high. The HOLD pin is sampled in each clock period in the HOLD state.
- The HOLD state terminates two clock periods after the HOLD signal goes low.

- During a six clock-period machine cycle if the HOLD signal is detected low at T₂, then it will be sampled again during T₄. If the HOLD is sampled high at T₄, then HOLD state will be initiated during T₆. Other timings will follow the same logic as HOLD in T₄ machine cycle described above.

3.6 EXTERNAL INTERRUPTS TIMING DIAGRAM

The 8085A has five interrupt request pins. These are TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR. The locations of Interrupt Servicing Routine (ISR) for all interrupts except INTR are fixed. Interrupt requests on TRAP, RST 7.5, RST 6.5 and RST 5.5 cause the 8085A to generate its own internal interrupt acknowledge instruction and branch to respective ISRs. This has already been explained previously.

When there is an interrupt request on INTR pin the 8085A checks for the following:

- (a) Whether any interrupt is being processed. This would disable all maskable interrupts unless enabled through EI instruction in ISR.
- (b) Whether the interrupt system is disabled (through DI instruction).
- (c) Whether any interrupt is pending on RST 7.5, RST 6.5, RST 5.5 or TRAP lines?

If all the above are not true, the 8085A accepts the interrupt request on completion of execution of the current instruction. The 8085A then executes an interrupt acknowledgement machine cycle (Figure 3.17) as follows:

- The 8085A samples INTR during the second last clock period of each instruction's execution.
- Even though memory is not being accessed, PC contents are put on the address bus during T₁. Since IO/M and RD signals are high, memory read is not performed. PC contents are also not incremented during the interrupt acknowledgement process. Thus, putting PC contents on the address bus does not cause any harm.
- The 8085A makes S₀, S₁ and IO/M high. These can be decoded by the interrupting device as advance acknowledgement. Note that S₀ = S₁ = High signifies an instruction fetch machine cycle, which is a memory operation. However, since IO/M is high it

signifies an I/O operation. This combination of signals which indicates an instruction fetch which $\overline{\text{INTA}}$ accesses I/O instead of memory, can be taken as interrupt acknowledgement.

- $\overline{\text{INTA}}$ goes low during T₂. The external logic must use both as device select signal and as a strobe signal to identify the time interval during which the interrupt acknowledge instruction code must be placed on the data bus.
- The external logic may respond to $\overline{\text{INTA}}$ signal by placing the Restart (RST) or Call instruction object code on the data bus.
- The instruction code is decoded in the subsequent clock cycles. If the instruction code corresponds to RST instruction, no further action is taken and the control branches to ISR addressed by the RST instruction in the next machine cycle.
- If the instruction code is for the “Call address” instruction (which is a 3-byte instruction), the interrupt acknowledgement extends to two more machine cycles.
- The second and third acknowledgement machine cycles are I/O read cycle, where IO/M and RD are output high and $\overline{\text{INTA}}$ is pulsed low.
- The external logic places two bytes of address (of ISR) on data bus in synchronization with $\overline{\text{INTA}}$, thus creating a Call instruction.

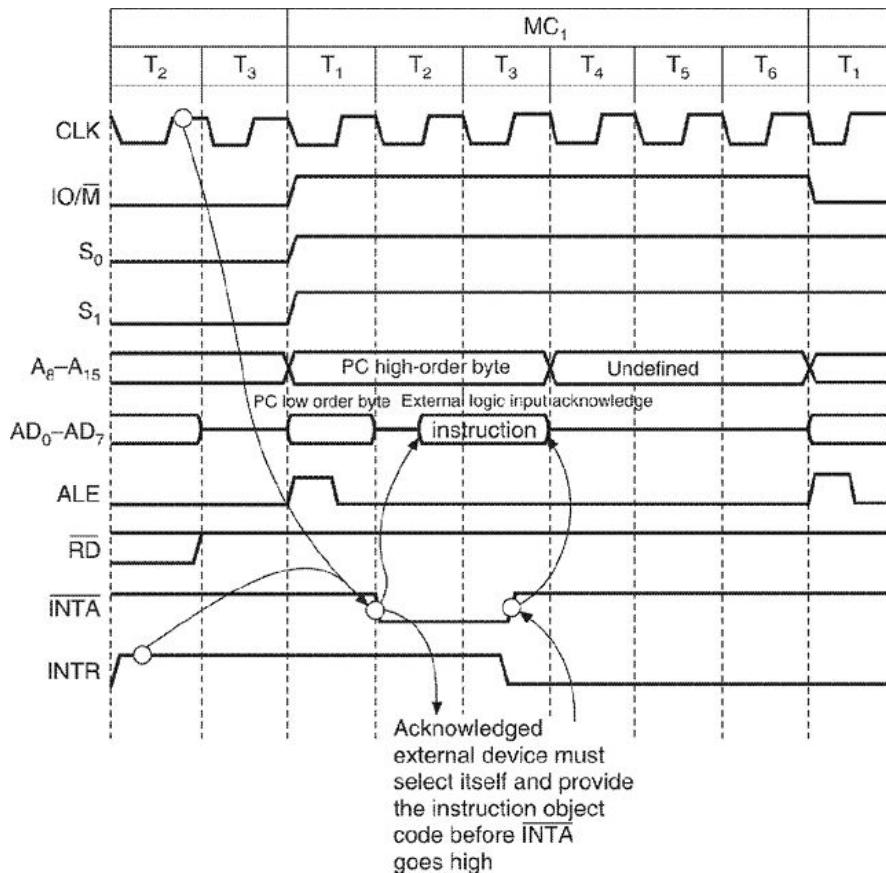


Figure 3.17 Interrupt acknowledgement through RST instruction.

The State Transition Diagram of the 8085, showing the sequence of all possible states, which the CPU can assume while executing an instruction has been shown in Chapter 1 at Figure 1.5.

3.7 CONCLUSION

In this chapter, we have learned about the architecture of the 8085 microprocessor, interfacing of main memory to microprocessor as well as how the microprocessor executes different instructions. In the next chapter we shall learn to program Intel 8085 to execute different tasks.

EXERCISES

1. The 8085A microprocessor needs to be interfaced to a 2 KB ROM chip and to three chips of 4 KB RAM. Draw the circuit diagram and explain the function.
2. The following settings for interrupts need to be made:
RST 7.5 : Disable
RST 6.5 : Enable

RST 5.5 : Enable

INTR : Enable

TRAP : Enable

Design the interrupt mask register bit pattern and write the assembly instructions to achieve the settings.

3. In the program flow given below, what will happen if during ISR 6.5, RST 7.5 interrupt occurs at Instruction MOV A, M?

MV1	A, 05
MOV	B, A
ADI	06
STA	2400H
MOV	C, A

Interrupt RST 6.5 occurs

—	ISR	6.5
—	LXI	H 2400H
—	MOV	A, M
—	ADI	05
—		
—		
	RET	

4. If in the above program, the RIM instruction is introduced after the instruction ADI 05, what will be the contents of register A at the conclusion of the RIM instruction. Remember that RST 7.5 has occurred at MOV A, M.

5. Draw the waveform indicating the variation of signal at the SOD line when the program given below is executed.

MVI A, B0H

SIM

MVI A, 00H

SIM

MVI A, FFH

SIM

MVI A, F0H

SIM

6. What are the utilities of S_0 and S_1 signals? Explain using a machine cycle diagram.

7. Draw the machine cycle for the execution of the following instructions:

(a) MVI A, 05

- (b) ADD B
(c) LX1 H, 2600H
(d) INR D
8. During the execution of a program, a programmed delay of 10 ms needs to be introduced between two instructions. Assume that the clock frequency of the 8085A is 5 MHz. Write the instruction block which may be incorporated between two instructions to introduce the delay.
 9. Draw the interrupt acknowledgement machine cycle with Call instruction object code being sent by the external logic in response to acknowledgement.
 10. Draw the I/O write machine cycle. Also draw the machine cycle for instruction ‘OUT Port’ Compare the two machine cycles.

FURTHER READING

Krishna Kant, *Microprocessor Based Data Acquisition System Design*, Tata McGraw-Hill, New Delhi, 1987.

Osborne Adam and Kane Jerry, *Some Real Microprocessors*, A. Osborne Associates Inc., Berkley, California, 1978.

Rafiquzzaman, Mohamed, *Microcomputer Theory and Applications with the Intel SDK-85*, John Wiley & Sons, 1982.

(Contd.)

4

INTEL 8085 MICROPROCESSOR INSTRUCTION SET AND PROGRAMMING

4.1 INTRODUCTION

The programming of a microprocessor is required to make it perform a required job. It may be a simple arithmetic calculation or a complex robot control. The ‘programming environment’ includes the hardware facilities (which can be accessed by the user), the Program Status Word, the operand types as well as the instruction set.

The hardware facilities of the 8085 have been amply described in Chapter 3. The following resources of the 8085 microprocessor are available to the user for programming.

Registers:	A (Accumulator), B, C, D, E, H, L
Register pairs: (16 bits)	BC, DE, HL In a register pair, the first register will be higher order while the second register will be lower order
PC:	16-bit Program Counter register
SP:	16-bit Stack Pointer register
PSW:	16-bit Program Status Word register
Condition Flags (1 bit):	Z (Zero), S (Negative Sign), CY (Carry), AC (Auxiliary Carry) and P (Even Parity)
Memory:	64 kB accessed through 16-bit address
I/O devices:	Total 256 input and 256 output ports accessed through 8-bit port address

Stack: Stack is part of memory and is accessed through 16-bit Stack Pointer (SP)

These facilities are mapped in microprocessor instructions, as we shall soon see. The above resources constitute the programmer's model of the 8085 (Figure 4.1).

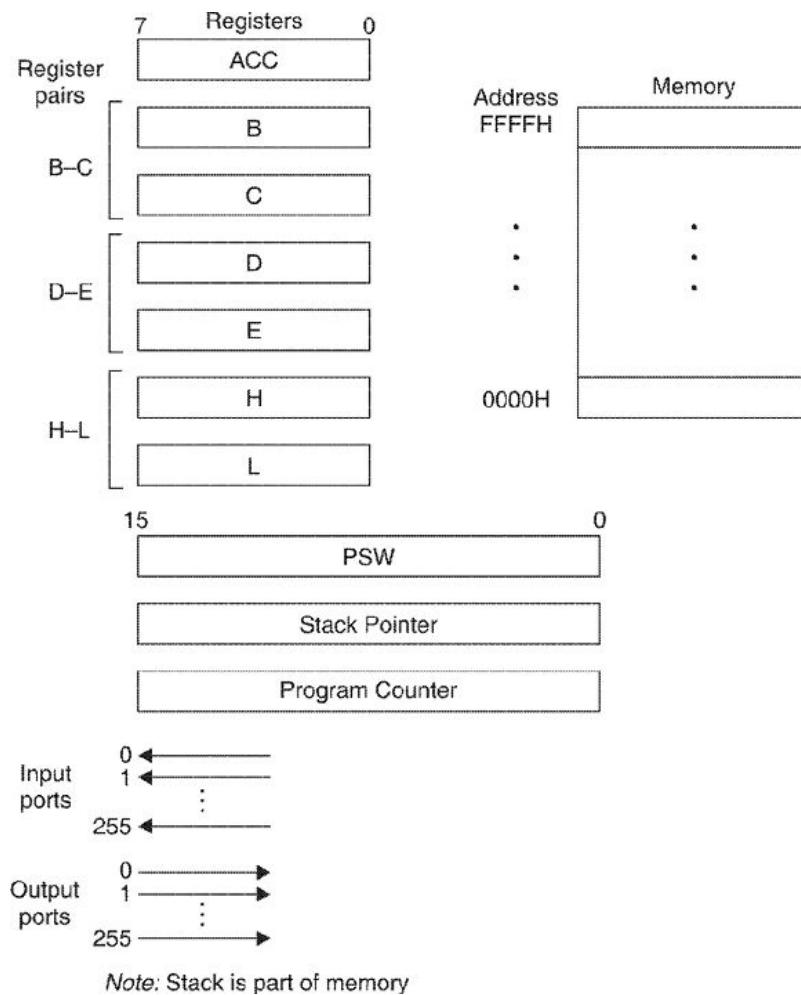


Figure 4.1 Programmer's model of Intel 8085.

4.2 PROGRAM STATUS WORD

The Program Status Word (PSW) is a 16-bit register. The higher-order 8 bits contain Accumulator contents and the lower-order 8 bits have five Condition Flags as shown below.

15	8	7	6	5	4	3	2	1	0
	ACC		S	Z	X	AC	X	P	X CY

PSW contents
X = Undefined

The PSW may be pushed on to stack to save the ACC and flags status during interrupt execution and subroutine execution.

The PUSH and POP instructions are used for storing and retrieving the PSW contents from the stack.

4.3 OPERAND TYPES

The 8085 is an 8-bit microprocessor. All calculations are performed on 8-bit data. If a programmer wants to perform 16-bit calculations, one has to do calculations on the lower-order byte followed by calculations on the higher-order byte.

4.4 INSTRUCTIONS FORMAT

The 8085 has single-byte, two-byte and three-byte instructions as shown in Figure 4.2. In case of two- or three-byte instructions, the lower-order byte/bytes contain data or address. Thus the first byte contains the operation code, while the other bytes contain the operand data or address, if required.

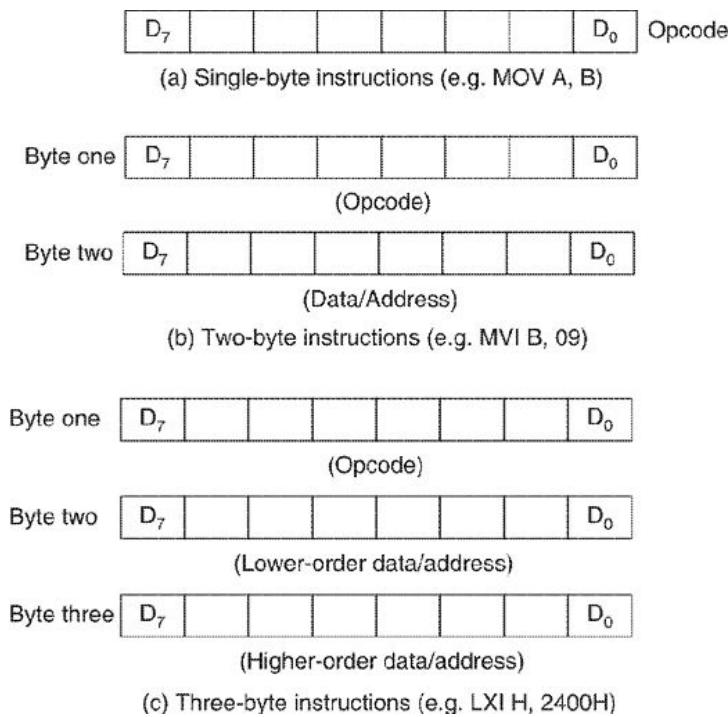


Figure 4.2 Instruction format.

4.5 ADDRESSING MODES

The addressing mode defines the way in which the operand of instructions can be accessed. The 8085 has four basic addressing modes to address the data stored in memory or register.

Immediate addressing

The data is stored along with the instruction. The data can be either 8 bit or 16 bit. In case of 16-bit data, the lower byte is stored first and the higher byte is stored next.

EXAMPLE 4.1

MVI A 0FH
(Load 0FH to register A)
Instruction in memory—

LXI D FFF0H
(Load FFF0H to D-E)

MVI A
0FH (data)

LXI D
F0H (lower byte)
FFH (higher byte)

After the execution of the above instructions, we get

- (A) = 0FH
 (D) (E) = FFF0H

Register addressing

The data is stored in a register or in the register pair specified.

EXAMPLE 4.2

MOV B, D (Move the contents of register D to register B)
INX H (Increment the contents of H-L register pair)

Originally,

(B) = 0FH (D) = 1FH
(H) = 0FH (L) = 1AH

After the execution of the above instructions,

(B) = 1FH (D) = 1 FH
(H)(L) = 0F1BH

Direct addressing

In this case, the data is stored in memory and the exact memory location is specified in the instruction as byte 2 (lower-order byte of address) and byte 3 (higher-order byte of address).

EXAMPLE 4.3

LDA 240FH (Load register A with the contents of memory location 240FH).

<i>Instruction in memory</i>	<i>Data in memory</i>
LDA	240EH
0FH	240FH
24H	2410H

After the execution of the above instruction,

$$(A) = 09H$$

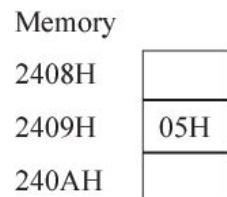
Register indirect addressing

The data is stored in memory and the address of memory location is stored in a register pair H–L, B–C, or D–E. The first register H, B or D contains the higher-order byte and the second register of register pair contains the lower-order byte of the address. The register pair H–L is widely used for the register indirect addressing. All the instructions, which involve the data transfer to/from memory without mentioning the register pair containing the address, take the address from the register pair H–L.

EXAMPLE 4.4

MOV B, M (Move the contents of memory location addressed by register pair H–L to register B).

$$(H) (L) = 2409H$$



After the execution of the above instruction,

$$(B) = 05H$$

There are two instructions (LDAX and STAX) for transfer of data from and to memory pointed by the address stored in register pairs B–C and D–E.

EXAMPLE 4.5

LDAX B (Move the contents of memory location addressed by register pair B–C to ACC).



After the execution of the above instruction,

$$(A) = FFH$$

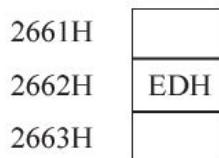
EXAMPLE 4.6

STAX D (Move the contents of register A to memory location addressed by register pair D–E).

(A) = EDH

(D) (E) = 2662H

After the execution of the above instruction



4.6 INSTRUCTION SET

4.6.1 Symbols and Abbreviations

The following symbols and abbreviations are used to describe the 8085 instructions.

Symbols	Meaning
Accumulator	Register A
r, r ₁ , r ₂	One of the registers A, B, C, D, E, H, L
rp	One of the register pairs
	B—register pair B–C
	D—register pair D–E
	H—register pair H–L
rh	High-order (first) register of designated register pair.
r _l	Low-order (second) register of a designated register pair.
SP	Stack Pointer
PC	Program Counter
rm	bit m of register r
addr	16-bit address
data	8-bit data
data 16	16-bit data
port	8-bit address of I/O device
byte 2	Second byte of instruction
byte 3	Third byte of instruction
Z, S, P, CY, AC	Condition Flags—Zero, Sign, Parity, Carry and Auxiliary Carry respectively.
()	The contents of memory location or register enclosed in parentheses
□	“are transferred to”
L	Logical And

V	Inclusive OR
⊻	Exclusive OR
+	Addition
-	Two's complement subtraction
□	Multiplication
□	“Is exchanged with”
-	One's complement (e.g. \bar{A})
n	The restart number (n = 0 to 7)

4.6.2 Data Transfer Instructions

Data transfer instructions are used to move data between registers, register pairs and between memory and registers. These instructions are described below.

MOV r1, r2 (Move Register)

(r1) □ (r2)

The contents of register r2 are moved to register r1.

Cycles: 1, States: 4, Addressing: Register, Flags: None.

MOV r, M (Move from Memory)

(r) □ ((H) (L))

The contents of the memory location, whose address is in registers H and L, are moved to register r.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: None.

MOV M, r (Move to Memory)

((H) (L)) □ (r)

The contents of register r are moved to the memory location whose address is in registers H and L.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: None.

MVI r, data (Move to Register Immediate)

(r) □ (byte 2)

The contents of byte 2 of the instruction are moved to register r.

Cycles: 2, States: 7, Addressing: Immediate, Flags: None.

MVI M, data (Move to Memory Immediate)

((H) (L)) □ (byte 2)

The contents of byte 2 of the instruction are moved to the memory location whose address is in registers H and L.

Cycles: 3, States: 10, Addressing: Immediate/Register Indirect, Flags: None.

LXI rp, data 16 (Load Register pair Immediate)

(rh) \square (byte 3)

(r1) \square (byte 2)

Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp. Byte 2 of the instruction is moved into the lower-order register (r1) of the register pair rp.

Cycles: 3, States: 10, Addressing: Immediate, Flags: None.

LDA addr (Load Accumulator Direct)

(A) \square ((byte 3) (byte 2))

The contents of the memory location, whose address is specified by byte 2 and byte 3 of the instruction, are moved to register A.

Cycles: 4, States: 13, Addressing: Direct, Flags: None.

STA addr (Store Accumulator Direct)

((byte 3) (byte 2)) \square (A)

The contents of the accumulator are moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.

Cycles: 4, States: 13, Addressing: Direct, Flags: None.

LHLD addr (Load H and L Direct)

(L) \square ((byte 3) (byte 2))

(H) \square ((byte 3) (byte 2) + 1)

The contents of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, are moved to register L. The contents of the memory location at the succeeding address are moved to register H.

Cycles: 5, States: 16, Addressing: Direct, Flags: None.

SHLD addr (Store H and L Direct)

((byte 3) (byte 2)) \square (L)

((byte 3) (byte 2) + 1) \square (H)

The contents of register L are moved to the memory location whose address is specified in byte 2 and byte 3. The contents of register H are moved to the succeeding memory location.

Cycles: 5, States: 16, Addressing: Direct, Flags: None.

LDAX rp (Load Accumulator Indirect)

(A) \square ((rp))

The contents of the memory location, whose address is in the register pair rp, are moved to register A. Note that only the register pair rp = B (registers B and C) or rp = D (registers D and E) may be specified.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: None.

STAX rp (Store Accumulator Indirect)

((rp)) □ (A)

The contents of register A are moved to the memory location whose address is in the register pair rp. Note that only the register pair rp = B (registers B and C) or the register pair rp = D (registers D and E) may be specified.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: None.

XCHG (Exchange H and L with D and E)

(H) □ (D)

(L) □ (E)

The contents of registers H and L are exchanged with the contents of registers D and E.

Cycles: 1, States: 4, Addressing: Register, Flags: None.

EXAMPLE 4.7

Four bytes of data are stored consecutively from location 2400H onwards. Write a program to transfer this data to start from 2500H onwards using (a) direct addressing (LDA), and (b) direct addressing (LHLD).

(a) LDA 2400H ; Load the contents of 2400H in ACC

STA 2500H ; Store the contents of ACC in 2500H

LDA 2401H

STA 2501H

LDA 2402H

STA 2502H

LDA 2403H

STA 2403H

(b) LHLD 2400H ; Load in L and H the contents of 2400H and 2401H

SHLD 2500H ; Store in 2500H and 2501H the contents of the L and H registers

LHLD 2402H

SHLD 2502H

EXAMPLE 4.8

Write a program to load registers A, B, C, D, and E with constant FFH.

MVI	A, FFH	; Move FFH to register A
MOV	B, A	; Move A to register B
MOV	C, A	
MOV	D, A	
MOV	E, A	

EXERCISES

1. What are the contents of registers A, B and C after the execution of the following instructions?

MVI	B, 0FH
MVI	C, FFH
MVI	A, 00H
STA	2400H
LXI	H, 2402H
MOV	M, C
MOV	A, B
LDA	2402H

2. The contents of memory locations 2400H, 2401H, 2402H and 2403H are 00, FFH, 0FH and 1AH respectively. What will be the memory contents after the following instructions are executed?

LHLD	2402H
LDA	2400H
MOV	B, L
MOV	C, H
MVI	D, 03
MOV	E, A
SHLD	2400H
STA	2402H

3. A value 05H is to be stored at memory locations 2000H, 2001H and 2002H. Write the different programs to perform this. Select the program that will take the least amount of memory and time to execute. Use only the instructions covered so far.

4.6.3 Arithmetic Instructions

These groups of instructions are used to perform arithmetic operations on data stored in registers and memory. The operations that can be performed are Add, Subtract, Increment, Decrement, etc. In case of add

and subtract, one of the operands is stored in register A and after the operation, the result is available in register A.

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Carry, and Auxiliary Carry flags according to the standard rules.

All subtraction operations are performed via two's complement arithmetic and set the carry flag to 1 to indicate a borrow and clear it to indicate no borrow.

The arithmetic instructions are given below:

ADD r (Add Register)

(A) \square (A) + (r)

The contents of register r are added to the contents of the accumulator. The result is placed in the accumulator.

Cycles: 1, States: 4, Addressing: Register, Flags: Z, S, P, CY, AC.

ADD M (Add Memory)

(A) \square (A) + ((H) (L))

The contents of the memory location, whose address is contained in the H and L registers, are added to the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

ADI data (Add Immediate)

(A) \square (A) + (byte 2)

The contents of the second byte of the instruction are added to the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

ADC r (Add Register with Carry)

(A) \square (A) + (r) + (CY)

The contents of register r and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

Cycles: 1, States: 4, Addressing: Register, Flags: Z, S, P, CY, AC.

ADC M (Add Memory with Carry)

(A) \square (A) + ((H) (L)) + (CY)

The contents of the memory location, whose address is contained in the H and L registers, and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

ACI data (Add Register with Carry)

(A) \square (A) + (byte 2) + (CY)

The contents of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

SUB r (Subtract Register)

(A) \square (A) - (r)

The contents of register r are subtracted from the contents of the accumulator. The result is placed in the accumulator.

Cycles: 1, States: 4, Addressing: Register, Flags: Z, S, P, CY, AC.

SUB M (Subtract Memory)

(A) \square (A) - ((H) (L))

The contents of the memory location, whose address is contained in the H and L registers, are subtracted from the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

SUI data (Subtract Immediate)

(A) \square (A) - (byte 2)

The contents of the second byte of the instruction are subtracted from the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

SBB r (Subtract Register with Borrow)

(A) \square (A) - (r) - (CY)

The contents of register r and the content of CY flag are both subtracted from the contents of the accumulator. The result is placed in the accumulator.

Cycles: 1, States: 4, Addressing: Register, Flags: Z, S, P, CY, AC.

SBB M (Subtract Memory with Borrow)

(A) \square (A) - ((H) (L)) - (CY)

The contents of the memory location, whose address is contained in the H and L registers, and the content of the CY flag, are both

subtracted from the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

SBI data (Subtract Immediate with Borrow)

(A) \square (A) – (byte 2) – (CY)

The contents of the second byte of the instruction and the content of the CY flag are both subtracted from the contents of the accumulator. The result is placed in the accumulator.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

INR r (Increment Register)

(r) \square (r) + 1

The contents of register r are incremented by one. Note that all condition flags except CY are affected.

Cycles: 1, States: 4, Addressing: Register, Flags: Z, S, P, AC.

INR M (Increment Memory)

((H) (L)) \square ((H) (L)) + 1

The contents of the memory location, whose address is contained in the H and L registers, are incremented by one. Note that all condition flags except CY are affected.

Cycles: 3, States: 10, Addressing: Register Indirect, Flags: Z, S, P, AC.

DCR r (Decrement Register)

(r) \square (r) – 1

The contents of register r are decremented by one. Note that all condition flags except CY are affected.

Cycles: 1, States: 4, Addressing: Register Flags: Z, S, P, AC.

DCR M (Decrement Memory)

((H) (L)) \square ((H) (L)) – 1

The contents of the memory location, whose address is contained in the H and L registers, are decremented by one. Note that all condition flags except CY are affected.

Cycles: 3, States: 10, Addressing: Register Indirect, Flags: Z, S, P, AC.

INX rp (Increment Register pair)

(rh) (r1) \square (rh) (r1) + 1

The contents of the register pair rp are incremented by one. Note that no condition flags are affected.

Cycles: 1, States: 6, Addressing: Register, Flags: None.

DCX rp (Decrement Register pair)

(rh) (r1) □ (rh) (r1) – 1

The contents of the register pair rp are decremented by one. Note that no condition flags are affected.

Cycles: 1, States: 6, Addressing: Register, Flags: None.

DAD rp (Add Register pair to H and L)

(H) (L) □ (H) (L) + (rh) (r1)

The contents of the register pair rp are added to the contents of the register pair H and L. The result is placed in the register pair H and L. Note that only the CY flag is affected. It is set if there is a carry out of the double precision add; otherwise it is reset.

Cycles: 3, States: 10, Addressing: Register, Flags: CY.

DAA (Decimal Adjust Accumulator)

The 8-bit number in the accumulator is adjusted to form two 4-bit Binary Coded Decimal (BCD) digits by the following process.

1. If the value of the least significant 4-bits of the accumulator is greater than 9 or if the AC flag is set, 6 is added to the accumulator.
2. If the value of the most significant 4-bits of the accumulator is now greater than 9, or if the CY flag is set, 6 is added to the most significant 4-bits of the accumulator.

Note that all flags are affected.

Cycles: 1, States: 4, Flags: Z, S, P, CY, AC.

EXAMPLE 4.9

Let us assume that two numbers X and Y, 8-bit each, are stored in memory locations 2400H and 2401H. Calculate Z1 and Z2 by the following equations and store the results in memory locations 2402H and 2403H.

$$Z1 = X + Y - 2$$

$$Z2 = X - Y + 10$$

LXI	H, 2400H	; Load registers H–L with 2400H
MOV	C, M	; Move X to register C

INX	H	; Increment register pair H-L. H-L
will now have 2401H		
MOV	D, M	; Move Y to register D
MOV	A, D	; Move Y to ACC
ADD	C	; Add Y to X, result in ACC
SUI	02	; Subtract 2 from X + Y
INX	H	; Increment H-L. H-L will have
2402H		
MOV	M, A	; Store (X + Y - 2) in 2402H
MOV	A, C	; Move X to ACC
SUB	D	; Calculate (X - Y) result in ACC
ADI	0AH	; Add 10
INX	H	; Increment H-L. H-L has 2403H
MOV	M, A	; Store (X - Y + 10) in 2403H

EXERCISES

1. Five memory locations 2401H, 2402H, 2403H, 2404H, and 2405H have some data items called X1, X2, X3, X4, and X5 respectively. Write a program to calculate and store in different memory locations as in the following:
 (2401H) X1 + X5
 (2402H) X5 - X2
 (2403H) X4 + X1 + X3
 (2404H) X4 - X1 - X2
 (2405H) X1 + X2 + X3 + X4
2. Write a program to transfer five bytes of data from 2400H onwards to 2500H using (a) H-L register-based indirect addressing, and (b) B-C or D-E register-based indirect addressing.
 Which program will take the least memory space and time to execute? How will you prove it?

4.6.4 Logical Instructions

These instructions are used to perform logical operations on data in registers and memory. The operations that can be performed are AND, OR, Exclusive OR, Rotate, Complement and Set.

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Auxiliary Carry, and Carry flags according to the standard rules.

The logic instructions are given below:

ANA r (AND Register)

(A) \square (A) L (r)

The contents of the register r are logically ANDed with the contents of the accumulator. The result is placed in the accumulator. The CY flag is cleared and AC is set.

Cycles: 1, States: 4, Addressing: Register Flags: Z, S, P, CY, AC.

ANA M (AND Memory)

(A) \square (A) L ((H) (L))

The contents of the memory location, whose address is contained in the H and L registers, are logically ANDed with the contents of the accumulator. The result is placed in the accumulator. The CY flag is cleared and AC is set.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

ANI data (AND Immediate)

(A) \square (A) L (byte 2)

The contents of the second byte of the instruction, are logically ANDed with the contents of the accumulator. The result is placed in the accumulator. The CY flag is cleared and AC is set.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

XRA r (Exclusive OR Register)

(A) \square (A) \vee (r)

The contents of the register r are Exclusive ORed with the contents of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

Cycles: 1, States: 4, Addressing: Register Flags: Z, S, P, CY, AC.

XRA M (Exclusive OR Memory)

(A) \square (A) \vee ((H) (L))

The contents of the memory location, whose address is contained in the H and L registers, are Exclusive ORed with the contents of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

XRI data (Exclusive OR Immediate)

(A) \square (A) \vee (byte 2)

The contents of the second byte of the instruction are Exclusive ORed with the contents of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

ORA r (OR Register)

(A) \square (A) V (r)

The contents of register r are Inclusive ORed with the contents of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

Cycles: 1, States: 4, Addressing: Register, Flags: Z, S, P, CY, AC.

ORA M (OR Memory)

(A) \square (A) V ((H) (L))

The contents of the memory location, whose address is contained in the H and L registers, is Inclusive ORed with the contents of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

ORI data (OR Immediate)

(A) \square (A) V (byte 2)

The contents of the second byte of the instruction are Inclusive ORed with the contents of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

CMP r (Compare Register)

(A) – (r)

The contents of register r are subtracted from the contents of the accumulator. The accumulator contents remain unchanged. The condition flags are set as a result of the subtraction. The Z flag is set to 1 if (A) = (r). The CY flag is set to 1 if (A) < (r).

Cycles: 1, States: 4, Addressing: Register, Flags: Z, S, P, CY, AC.

CMP M (Compare Memory)

(A) – ((H) (L))

The contents of the memory location, whose address is contained in the H and L registers, are subtracted from the contents of the accumulator. The accumulator contents remain unchanged. The condition flags are set as a result of the subtraction. The Z flag is set to 1 if $(A) = ((H)(L))$. The CY flag is set to 1 if $(A) < ((H)(L))$.

Cycles: 2, States: 7, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

CPI data (Compare Immediate)

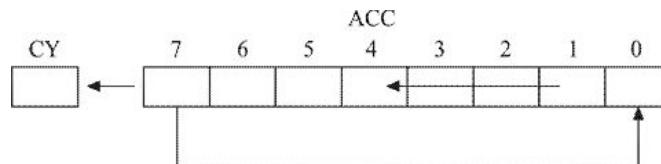
$(A) - (\text{byte } 2)$

The contents of the second byte of the instruction are subtracted from the contents of the accumulator. The accumulator contents remain unchanged. The condition flags are set by the result of the subtraction. The Z flag is set to 1 if $(A) = (\text{byte } 2)$. The CY flag is set to 1 if $(A) < (\text{byte } 2)$.

Cycles: 2, States: 7, Addressing: Immediate, Flags: Z, S, P, CY, AC.

RLC (Rotate Left)

$(A_{n+1}) \square (A_n); (A_0) \square (A_7); (\text{CY}) \square (A_7)$

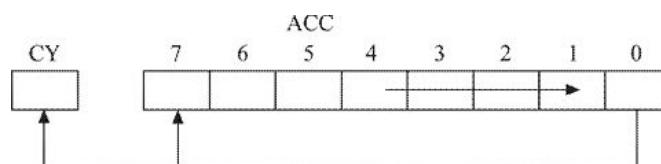


The contents of the accumulator are rotated left one position. The least significant bit (LSB) and the CY flag are both set to the value shifted out of the most significant bit (MSB) position. Only the CY flag is affected.

Cycles: 1, States: 4, Flags: CY.

RRC (Rotate Right)

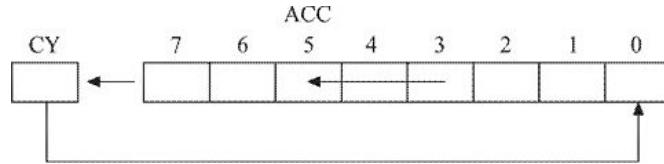
$(A_{n-1}) \square (A_n); (A_7) \square (A_0); (\text{CY}) \square (A_0)$



The contents of the accumulator are rotated right one position. The most significant bit (MSB) and the CY flag are both set to the value shifted out of the least significant bit (LSB) position. Only the CY flag is affected.

Cycles: 1, States: 4, Flags: CY.

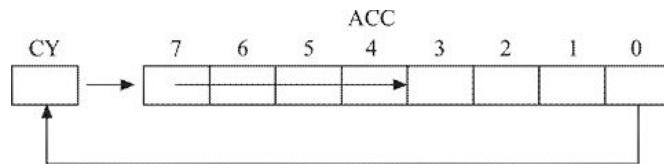
RAL (Rotate Left through Carry)
 $(A_{n+1}) \square (A_n); (A_0) \square (CY); (CY) \square (A_7)$



The contents of the accumulator are rotated left one position through the CY flag. The least significant bit (LSB) is set equal to the CY flag and the CY flag is set to the value shifted out of the most significant bit (MSB). Only the CY flag is affected.

Cycles: 1, States: 4, Flags: CY.

RAR (Rotate Right through Carry)
 $(A_{n-1}) \square (A_n); (A_7) \square (CY); (CY) \square (A_0)$



The contents of the accumulator are rotated right one position through the CY flag. The most significant bit (MSB) is set to the CY flag and the CY flag is set to the value shifted out of the least significant bit (LSB) position. Only the CY flag is affected.

Cycles: 1, States: 4, Flags: CY.

CMA (Complement Accumulator)
 $(A) \leftarrow (\bar{A})$

The contents of the accumulator are complemented (0 bit becomes 1, 1 bit becomes 0). No flags are affected.

Cycles: 1, States: 4, Flags: None.

CMC (Complement Carry)
 $(CY) \leftarrow (\bar{CY})$

The CY flag is complemented. No other flag is affected.

Cycles: 1, States: 4, Flags: CY.

STC (Set Carry)
 $(CY) \square 1$

The CY flag is set to 1. No other flag is affected.

Cycles: 1, States: 4, Flags: CY.

EXAMPLE 4.10

An 8-bit number X is stored in memory location 2400H. Write a program to compute Z by the following relation and store it in memory location 2401H.

$$Z = [\{ (X + 5) \text{ AND } (X - 5) \} \text{ OR } (X \square 2)]$$

The program to compute Z by the above relation is as follows:

LXI	H, 2400H	
MOV	A, M	; (A) \square X
ADI	05H	
MOV	B, A	; (B) \square X + 5
MOV	A, M	
SUI	05	; (A) \square X - 5
ANA	B	
MOV	E, A	; (E) \square (A) AND (B)
MOV	A, M	
ADD	A	; (A) \square X \square 2
ORA	E	; (A) \square (E) OR (A)
INX	H	
MOV	M, A	; (2401H) \square (A)

EXERCISES

- Three numbers X₁, X₂ and X₃ are stored at 2000H, 2001H and 2002H respectively. Write programs to calculate Z₁, Z₂ and Z₃ as per the following and store the result at 2100H, 2101H and 2102H respectively.

$$Z_1 = (X_1 \text{ OR } X_2) \text{ AND } X_3$$

$$Z_2 = (X_1 + X_3) \text{ XOR } X_2$$

$$Z_3 = (X_3 - X_1) \text{ AND } X_2$$

- Two numbers N₁ and N₂ are stored in registers B and C respectively. Write a program to calculate N₃ as per the following relation and store the result in register D.

$$N_3 = (N_1 \square 8) \text{ AND } (N_2 / 16)$$

- What will be the values in registers A, H, L and locations 2400H, 2401H, 2500H and 2501H after the execution of the following instructions?

LXI H, 2400

MVI A, 08H

MOV M, A

INX	H
SUI	02H
MOV	M, A
LHLD	2400H
SHLD	2500H
LDA	2501H
RLC	
RLC	
RLC	
ANI	98H

4.6.5 Branch Instructions

This refers to the group of instructions that alters the normal sequential program flow. There are two types of branch instructions—unconditional and conditional.

In case of unconditional branch, the branch address (i.e. the address of the memory location to which the jump is made) is loaded into the program counter (PC), whereas in case of conditional branch, the branch address is loaded into the PC, only if the specified condition is satisfied. The status of each condition flag is checked in order to determine whether the condition is satisfied or not.

Condition	Flag
NZ — Not Zero	(Z) = 0
Z — Zero	(Z) = 1
NC — No carry	(CY) = 0
C — Carry	(CY) = 1
PO — Odd Parity	(P) = 0
PE — Even Parity	(P) = 1
P — Positive Number	(S) = 0
M — Minus (Negative number)	(S) = 1

None of the flags are affected by the branch instructions. The branch instructions are explained below:

JMP addr (Jump)

(PC) □ (byte 3) (byte 2)

Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

Cycles: 3, States: 10, Addressing: Immediate, Flags: None.

J Condition addr (Conditional Jump)

If Condition = True, then

(PC) \square (byte 3) (byte 2)

else (PC)

If the specified condition is true, the control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction; otherwise, the control continues sequentially.

Cycles: 2/3, States: 7/10, Addressing: Immediate, Flags: None.

PCHL (Jump H and L Indirect – Move H and L to PC)

(PCH) \square (H)

(PCL) \square (L)

The contents of register H are moved to the high-order 8 bits of the register PC. The contents of register L are moved to the low-order 8 bits of the register PC.

Cycles: 1, States: 6, Addressing: Register, Flags: None.

Subroutine call and return instructions

There are other types of branch instructions like CALL, C.Condition, RET, R. Condition, and RST. These instructions allow the program to jump to a certain place to execute a number of instructions and then return to the same original place. This is made possible by storing the address of memory location from where the jump was made (called return address) in stack.

EXAMPLE 4.11

Assume that two numbers X and Y are stored in memory. Write a program to calculate Z by the following equation and store in memory.

$$Z = (X + Y) \square 30 - X \square Y \square 3$$

The flowchart for solving this problem is shown in Figure 4.3.

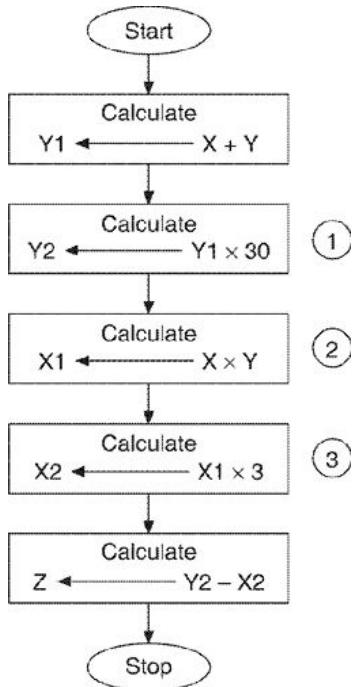


Figure 4.3 Flowchart to solve equation: $Z = (X + Y) \times 30 - X \times Y / 3$.

We know that the 8085 does not have the multiplication and division instructions, so these operations should be performed by the programmer using one of the existing techniques. The multiplication is to be performed at three different places—(1), (2) and (3) as shown in Figure 4.3. If the multiplication program takes 20 bytes of memory, 60 bytes in the program will be utilized in the multiplication operation only, whereas most of the instructions apart from loading the operands and storing the results are going to be the same. Thus, it is a waste of memory and the programmer's time as well.

Obviously, a better approach will be to store the instructions concerning the multiplication at a separate place in memory and execute them whenever multiplication is desired. Clearly, to make this successful, it is very necessary that the program returns to the same place, from where it branched to execute multiplication. Thus, the return address should be stored and fetched back in PC when the multiplication is over. This is achieved by using the CALL and RETURN instructions. The program segment, where the program branches in response to CALL instruction to perform specified operations, is called the **subroutine**. In our example, the multiplication program is called the subroutine. The stack is used to store the **return** address. The structure of CALL and RETURN is explained in Figure 4.4.

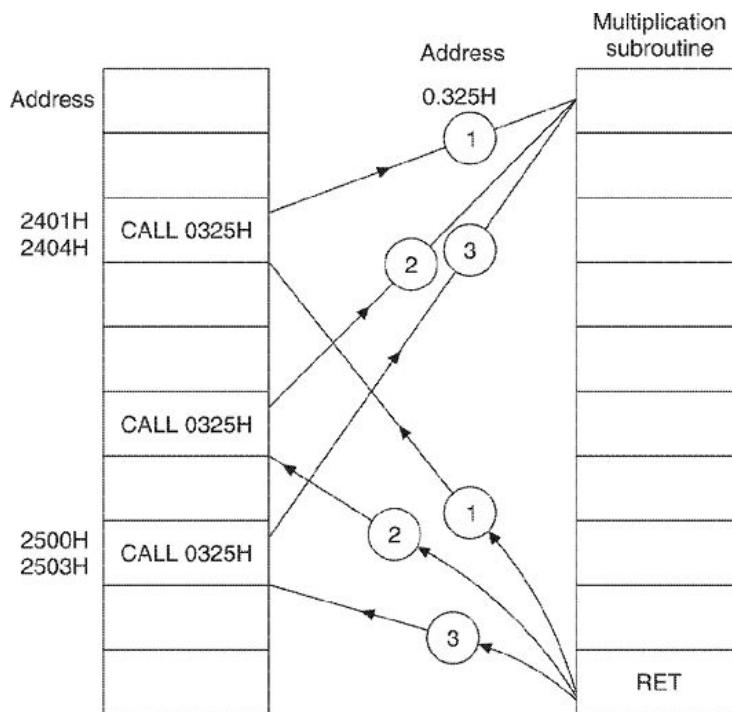


Figure 4.4 Structure of CALL and RETURN instructions.

The C.Condition and R.Condition instructions perform Call and Return operations respectively when the specified conditions are satisfied.

Following are the instructions:

CALL addr (Call)

$((SP) - 1) \square (PCH)$

$((SP) - 2) \square (PCL)$

$(SP) \square (SP) - 2$

$(PC) \square (\text{byte } 3) (\text{byte } 2)$

The high-order 8 bits of the next instruction address are moved to the memory location whose address is 1 less than the contents of register SP. The low-order 8 bits of the next instruction address are moved to the memory location whose address is 2 less than the contents of register SP. The contents of register SP are decremented by 2. The control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

Cycles: 5, States: 18, Addressing: Immediate/Register Indirect, Flags: None.

C Condition addr (Conditional Call)

If Condition = True, then

$((SP) - 1) \square (PCH)$

$((SP) - 2) \square (PCL)$

$(SP) \square (SP) - 2$
 $(PC) \square (\text{byte } 3) (\text{byte } 2)$
else (PC)

If the specified condition is true, the actions specified in the CALL instruction are performed; otherwise, the control continues sequentially.

Cycles: 2/5, States: 9/18, Addressing: Immediate/Register Indirect, Flags: None.

RET (Return)
 $(PCL) \square ((SP));$
 $(PCH) \square ((SP) + 1);$
 $(SP) \square (SP) + 2$

The contents of the memory location, whose address is specified in register SP, are moved to the low-order 8 bits of register PC. The contents of the memory location, whose address is 1 more than the contents of register SP, are moved to the high-order 8 bits of register PC. The contents of register SP are incremented by 2.

Cycles: 3, States: 10, Addressing: Register Indirect, Flags: None.

R Condition (Conditional Return)
If Condition = True, then
 $(PCL) \square ((SP))$
 $(PCH) \square ((SP) + 1)$
 $(SP) \square (SP) + 2$
else (PC)

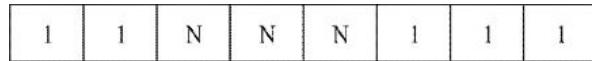
If the specified condition is true, the actions specified in the RET instruction are performed; otherwise, the control continues sequentially.

Cycles: 1/3, States: 6/12, Addressing: Register Indirect, Flags: None.

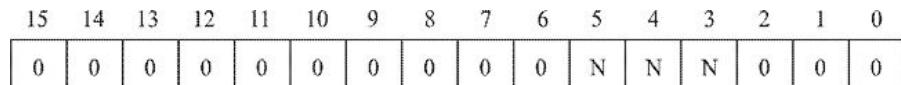
RST n (Restart)
 $((SP) - 1) \square (PCH)$
 $((SP) - 2) \square (PCL)$
 $(SP) \square (SP) - 2$
 $(PC) \square 8 \square n$

The high-order 8 bits of the next instruction address are moved to the memory location whose address is 1 less than the contents of register SP. The low-order 8 bits of the next instruction address are moved to the memory location whose address is 2 less than the contents of register SP. The contents of register SP are decremented by 2. The

control is transferred to the instruction whose address is 8 times the contents of NNN (n is represented as a 3-bit number NNN in opcode). The opcode format is



The Program Counter after the execution of RST n will be



Cycles: 3, States: 12, Addressing: Register Indirect, Flags: None.

EXAMPLE 4.12

A number X is stored in a memory location. Write a program to calculate the factorial X and store it in memory.

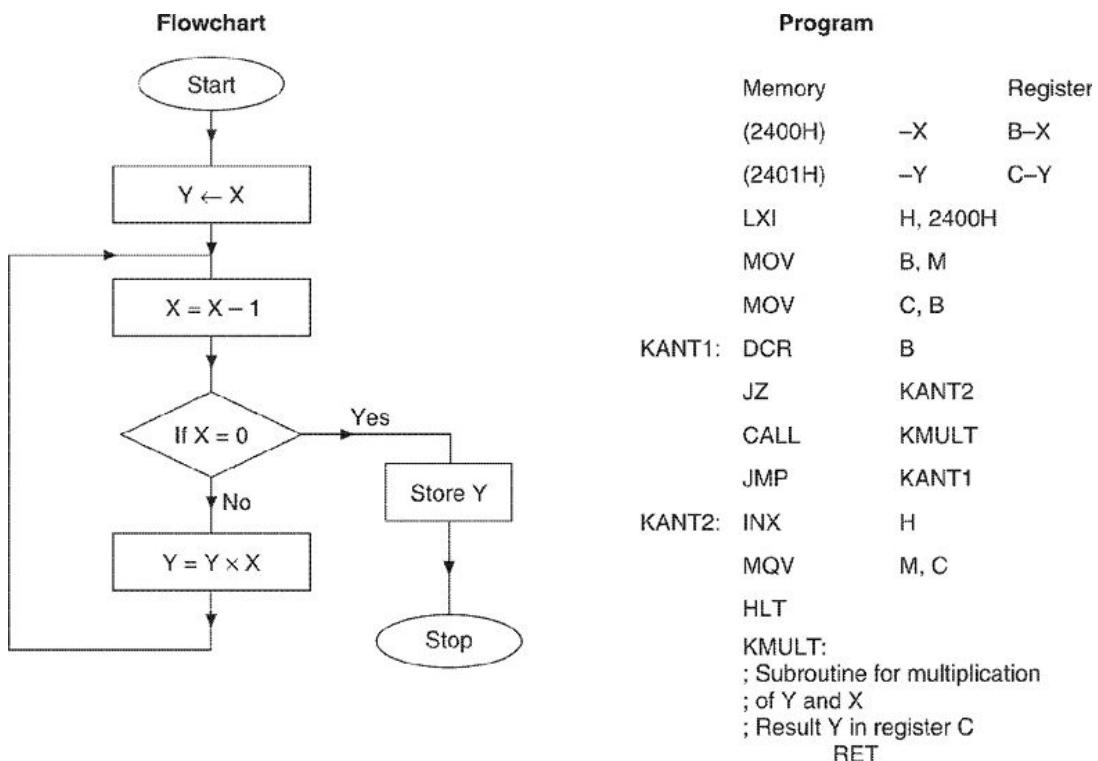


Figure 4.5 Flowchart to calculate factorial X.

The flowchart and the program are shown in Figure 4.5. This example illustrates the conditional jump, the unconditional jump, the subroutine CALL and the RETURN instructions and their utility in solving problems. The reader can easily write the KMULT subroutine for multiplication and complete the program.

EXERCISES

1. An array of 20 numbers is stored in memory starting from 2400H. A number X is stored in location 2500H. Write a program to search the number in the array and store the number of occurrences of the number in 2501H.
2. An array of 20 numbers is stored in memory starting from 2400H. Write a program to find the maximum number in the array.
3. Modify the program written in Exercise 2 to find the minimum number in the array. How many instructions did you change?
4. An array of 20 numbers in ascending order is stored in memory starting from 2400H. Write programs to search a number X by sequential search and binary search. Compare the two search strategies in terms of the number of searches required.

4.6.6 Stack I/O and Machine Control Instructions

These instructions are used to manipulate the stack, to perform the input–output and to alter the internal control flags. Unless otherwise specified, the condition flags are not affected by any instruction in this group. The instructions in this group are as follows:

PUSH rp (Push)

$((SP) - 1) \square (rh)$

$((SP) - 2) \square (rl)$

$(SP) \square (SP) - 2$

The contents of the high-order register of register pair rp are moved to the memory location whose address is 1 less than the contents of register SP. The contents of the low-order register of register pair rp are moved to the memory location whose address is 2 less than the contents of register SP. The contents of register SP are decremented by 2. Note that the register pair rp = SP may not be specified.

Cycles: 3, States: 12, Addressing: Register Indirect, Flags: None.

PUSH PSW (Push Program Status Word)

$((SP) - 1) \square (A)$

$((SP) - 2)_0 \square (CY), ((SP) - 2)_1 \square X$

$((SP) - 2)_2 \square (P), ((SP) - 2)_3 \square X$

$((SP) - 2)_4 \square (AC), ((SP) - 2)_5 \square X$

$((SP) - 2)_6 \square (Z), ((SP) - 2)_7 \square (S)$

$(SP) \square (SP) - 2 X$: Undefined

The contents of register A and the contents of condition flags which form the Program Status Word (PSW) are pushed on to stack.

The contents of register A are moved to the memory location whose address is 1 less than contents of register SP. The contents of the condition flags are assembled as in lower byte of Program Status Word and the byte is moved to the memory location whose address is 2 less than the contents of register SP. The contents of register SP decremented by 2.

Cycles: 3, States: 12, Addressing: Register Indirect, Flags: None.

POP rp (Pop)

(rl) $\square ((SP))$

(rh) $\square ((SP) + 1)$

(SP) $\square ((SP) + 2)$

The contents of the memory location, whose address is specified by the contents of register SP, are moved to the low-order register of register pair rp. The contents of the memory location, whose address is 1 more than the contents of register SP, are moved to the high-order register of register pair rp. The contents of register SP are incremented by 2. Note that the register pair rp = SP may not be specified.

Cycles: 3, States: 10, Addressing: Register Indirect, Flags: None.

POP PSW (Pop Program Status Word)

(CY) $\square ((SP))_0$, (P) $\square ((SP))_2$

(AC) $\square ((SP))_4$, (Z) $\square ((SP))_6$

(S) $\square ((SP))_7$

(A) $\square ((SP) + 1)$

(SP) $\square (SP) + 2$

The contents of the memory location, whose address is specified by the contents of register SP are used to restore the condition flags. The contents of the memory location whose address is 1 more than the contents of register SP are moved to register A. The contents of register SP are incremented by 2.

Cycles: 3, States: 10, Addressing: Register Indirect, Flags: Z, S, P, CY, AC.

XTHL (Exchange Stack top with H and L)

(L) $\square ((SP))$

(H) $\square ((SP) + 1)$

The contents of the register L are exchanged with the contents of the memory location whose address is specified by the contents of register SP. The contents of the register H are exchanged with the contents of the memory location whose address is 1 more than the contents of register SP.

Cycles: 5, States: 16, Addressing: Register Indirect, Flags: None.

SPHL (Move HL to SP)

(SP) □ (H) (L)

The contents of registers H and L (16 bits) are moved to register SP.

Cycles: 1, States: 6, Addressing: Register, Flags: None.

IN port (Input)

(A) □ (data)

The data placed on the 8-bit bidirectional data bus by the specified port is moved to register A.

Cycles: 3, States: 10, Addressing: Direct, Flags: None.

OUT port (Output)

(data) □ (A)

The contents of register A are placed on the 8-bit bidirectional data bus for transmission to the specified port.

Cycles: 3, States: 10, Addressing: Direct, Flags: None.

EI (Enable Interrupts)

The interrupt system is enabled following the execution of the next instruction.

Cycles: 1, States: 4, Flags: None.

DI (Disable Interrupts)

The interrupt system is disabled immediately following the execution of the DI instruction.

Cycles: 1, States: 4, Flags: None.

HLT (Halt)

The processor is stopped. The registers and flags are unaffected.

Cycles: 1, States: 5, Flags: None.

NOP (No Operation)

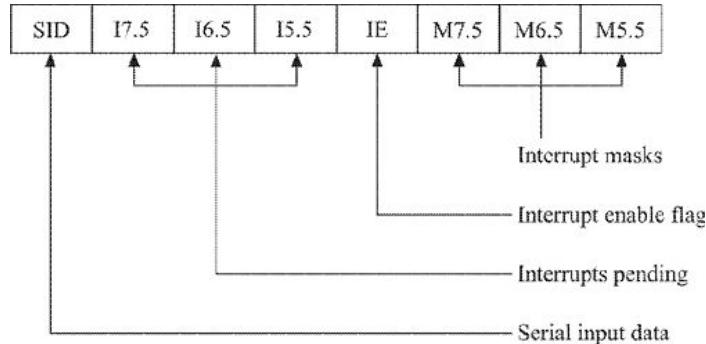
No operation is performed. The registers and flags are unaffected.

Cycles: 1, States: 4, Flags: None.

RIM (Read Interrupt Mask)

After the execution of the RIM instruction, the accumulator is loaded with the restart interrupt masks, pending interrupts if any, and the contents of the serial input data line (SID).

The accumulator contents after the execution of RIM instruction will be



Cycles: 1, States: 4, Flags: None.

SIM (Set Interrupt Masks)

During the execution of the SIM instruction, the contents of the accumulator will be used in programming the restart interrupt masks. Bits 0–2 will set/reset the mask bit for RST 5.5, 6.5, 7.5 of the interrupt mask register, if the bit 3 is 1 (“set”). Bit 3 is a “Mask Set Enable” control.

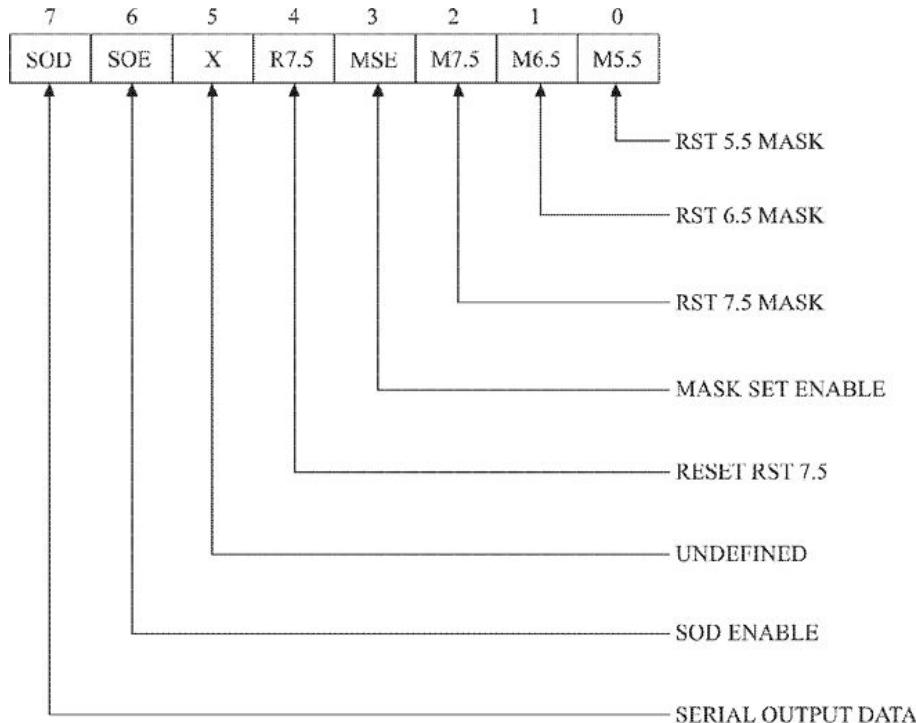
Setting the mask (i.e. mask bit = 1) disables the corresponding interrupt.

	<i>Set</i>	<i>Reset</i>
RST 5.5 Mask	if bit 0 = 1	if bit 0 = 0
RST 6.5 Mask	bit 1 = 1	bit 1 = 0
RST 7.5 Mask	bit 2 = 1	bit 2 = 0

RST 7.5 (edge trigger enable) internal request flip-flop will be reset if the bit 4 of the accumulator = 1, regardless of whether RST 7.5 is masked or not. RESETIN input (pin 30) will set all RST Masks, and reset/disable all interrupts.

SIM can also load the SOD output latch. Accumulator bit 7 is loaded into the SOD latch if the bit 6 is set. The latch is unaffected if the bit 6 is zero. RESETIN input sets the SOD latch to zero.

The accumulator contents for the SIM instruction are



Cycles: 1, States: 4, Flags: None.

4.7 CONCLUSION

The 8085, though considered as the first generation microprocessor, is still quite popular and widely used. It is also taught in all microprocessor first-level courses because of its powerful addressing modes and instruction set. Another reason is the availability of a large number of support chips which can be used to interface the 8085 to different I/O devices. This will be evident in Chapter 7 on Microprocessor Peripheral Interfacing.

EXERCISES

Write programs to solve the following problems.

1. After the execution of RIM instruction the accumulator content is 77H. What information does it convey?
2. Plot the voltage level at the SOD pin of the 8085 when the following instructions are executed.

```

MVI      A, FFH
SIM
MVI      A, 8FH
SIM
MVI      A, C0H
  
```

SIM

MVI A, 80H

3. What happens when an RST 6.5 interrupt occurs at LDA 2400H?

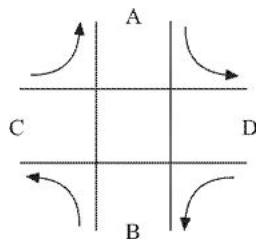
MVI A, F1H
MOV B, A
LXI H, 2400H
MOV M, B
ADI 05
MVI A, 07
SIM
LDA 2400H
MOV D, A

4. An 8-bit number X is stored in memory. Check the fourth bit of X. Store $(X + 6)$ as result in some memory location if the specified bit is 1; otherwise store $(X - 6)$.
5. Two numbers X and Y are stored in memory. Calculate the value of the expression $(X + Y - 100)$. If the value is negative, calculate the value of Z_1 and store; otherwise, calculate the value of Z_2 and store in memory. The equations pertaining to Z_1 and Z_2 are as follows:

$$Z_1 = (X \text{ AND } Y) \text{ OR } ((X - 5) \text{ OR } (Y - 5))$$

$$Z_2 = ((X - 5) \text{ AND } (Y - 5)) \text{ OR } (X - Y).$$

6. Multiply two 8-bit numbers stored in memory and store the 16-bit result in memory.
7. Divide two 8-bit numbers X and Y and store the result in memory.
8. A block of 32 bytes is stored in consecutive locations in memory. Move this block to some other portion of memory.
9. Ten numbers are stored in consecutive memory locations. Arrange them in ascending order.
10. A number X is stored in memory. Calculate $\exp(X)$ and store in 16 bits. Ignore fractions during divisions and go up to X^5 only.
The expression for e^X is
- $$e^X = 1 + \frac{X}{1} + \frac{X^2}{2} + \frac{X^3}{3} + \frac{X^4}{4} + \dots$$
11. In this problem, you have to simulate a traffic light system on a crossing of roads. Let us identify the roads as A, B, C and D.



When the traffic on road AB is cleared, we signify this event ABBA meaning that traffic from A to B and B to A is cleared. In the same way, we can signify the other road clearing events as BDAC, CDDC and CBDA. The sequence of these events will be ABBA, BDAC, CDDC, CBDA and ABBA again. The time between two events is taken as about 10 seconds. Use memory locations for events.

FURTHER READING

- Eventual, Lance A., *8080/8085A Assembly Level Programming*, A. Osborne McGraw-Hill, Berkeley, California, 1978.
- Krishna Kant, *Microprocessor Based Data Acquisition System Design*, Tata McGraw-Hill, New Delhi, 1987.
- MCS 85 Users Manual*, Intel Corporation, Santa Clara, 1983.
- Rafiquzzaman, Mohamed, *Microcomputer Theory and Applications with the Intel SDK 85*, John Wiley & Sons, 1982.

5

INTEL 8086 HARDWARE ARCHITECTURE

5.1 INTRODUCTION

The 8086 is the Intel's first 16-bit microprocessor. Implemented in n -channel, depletion mode silicon gate technology and packaged in 40 pin dual-in-line package, it was significantly more powerful than any previous microprocessor. The 8086 assembly language instruction set is upward compatible at source level with that of the 8080A. Thus it is possible to convert the 8080A assembly language instruction into one or more of the 8086 assembly level instructions.

The 8086 architecture illustrates the implementation of two-stage pipelining in instruction execution (refer Chapter 2 for details on pipelining). The central processing logic has been divided into an Execution Unit (EU) and a Bus Interface Unit (BIU). These units operate asynchronously. The Bus Interface Unit provides interface with external bus, and executes all bus operations. It has a 6-byte instruction object code queue. The Execution Unit, on the other hand, takes the instruction from object code queue of BIU and executes it. Thus, the instruction fetch-time has been largely eliminated.

The 8086 has been designed keeping in mind the number of application environments ranging from a single microprocessor to multi-microprocessor systems. To accommodate this requirement the 8086 has two operating modes—minimum and maximum. The minimum mode of operation caters to a single-processor environment whereas the maximum mode is for multiprocessor environment. The mode selection is carried out by applying high-level or low-level at the $\overline{MN/MX}$ pin. A number of the 8086 pins output alternate signals, depending on the voltage level at the $\overline{MN/MX}$ pin, to support the flexibility in operation. The 8086 system memory has four segments—Code Segment, Data Segment, Stack Segment and Extra Segment. Each segment can have a

maximum capacity of 64 KB. The starting addresses of segments in memory are given by the segment registers. This enables program relocation as described briefly in Chapter 2.

The 8086 and 8088 have the same instruction set and differ very little in their architecture. The descriptions in this chapter therefore relate to both 8088 and 8086; the differences between them, if any, will be specifically mentioned as we go along.

The 8086 works with a number of support chips such as

- 8284 Clock Generator/Driver to provide clock signal
- 8288 Bus Controller (usually used in a multiprocessor environment).

5.2 ARCHITECTURE

The 8086 is a 16-bit microprocessor with 20 bit address bus and 16 bit data bus. Thus it can directly access $2^{20} = 1,048,567$ (1 MB) memory locations and can read/write 8 bits or 16 bits data from/to memory or I/O. Like 8085, the 8086 also has multiplexed address and data buses, which reduces the number of pins needed. However, it also slows down the data transfer rate. The 8088 has an 8-bit data bus. Thus, only 8-bit read/write at one time is possible.

The 8086 was announced with 5 MHz clock. However, the later versions Intel 8086-2 and Intel 8086-1 are available with 8 MHz and 10 MHz clocks respectively. The 8086 requires a single-phase clock with 33% duty cycle to provide internal timing.

The internal architecture of the 8086 is shown in Figure 5.1. It is divided into two separate functional units—Bus Interface Unit (BIU) and Execution Unit (EU). These two units work simultaneously for instruction execution and form two-stage instruction pipeline.

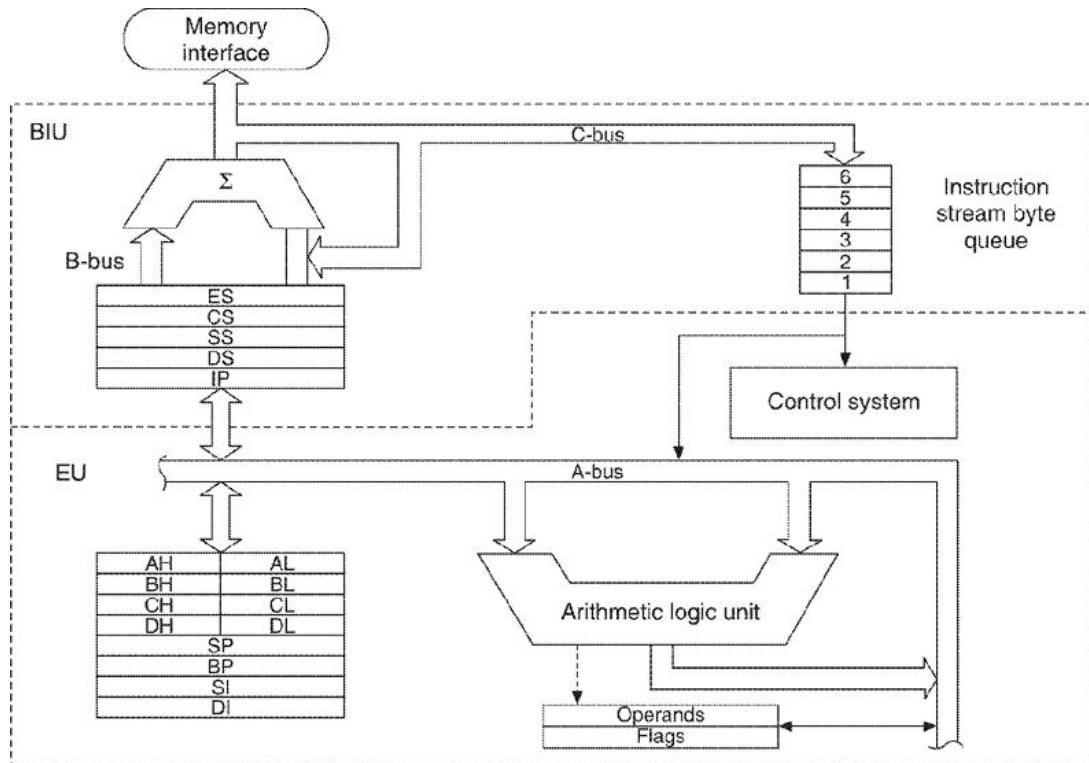


Figure 5.1 Intel 8086 internal architecture.

5.2.1 Bus Interface Unit (BIU)

The BIU contains Bus Interface Logic, Segment Registers, Memory Addressing Logic and a 6-byte Instruction Object Code Queue. The BIU performs all bus operations for the Execution Unit, and is responsible for executing all external bus cycles. When the EU is busy in instruction execution, the BIU continues fetching instructions from memory and stores them in the instruction queue.

If the EU executes an instruction, which transfers the control of the program to another location, then the BIU

- resets the queue,
- fetches the instruction from the new address,
- passes the instructions to the EU, and
- begins refilling the queue from the new location.

As described in Chapter 2, this process is called pipeline flushing.

The BIU of the 8088 has the same architecture except that it has 4-byte instruction queue.

Each time there is one byte hole in the queue, the 8088 BIU fetches a new instruction byte to load into the queue. On the other hand, the 8086 BIU fills the queue when the queue contains empty spaces of two bytes.

5.2.2 Execution Unit

The Execution Unit (EU) contains the complete infrastructure required to execute an instruction, i.e. Instruction Decoder, Arithmetic Logic Unit, General Purpose Registers, Pointers and Index Registers, Flag Register and Control Circuitry.

The EU is responsible for

- the execution of all instructions,
- providing address to the BIU for fetching data/instruction, and
- manipulating various registers as well as the flag register.

Except for a few control pins, the EU is completely isolated from the outside world and it is the BIU which performs all the bus operations for the EU.

We dealt with pipelining in detail in Chapter 2. Therefore, we shall not describe it in this chapter. The overlapped fetching and execution of instructions in the 8086 is shown in Figure 5.2.

The numbers above the operation boxes in Figure 5.2 show the instruction number in the instruction stream. Considering that a second generation microprocessor completes the operation in the same number of clock cycles as the 8086/8088 microprocessor, it shows that due to overlapping of fetch and execute operations, the 8086/8088 is not only able to execute three instructions in lesser time but also able to complete the fetching of fourth and fifth instructions.

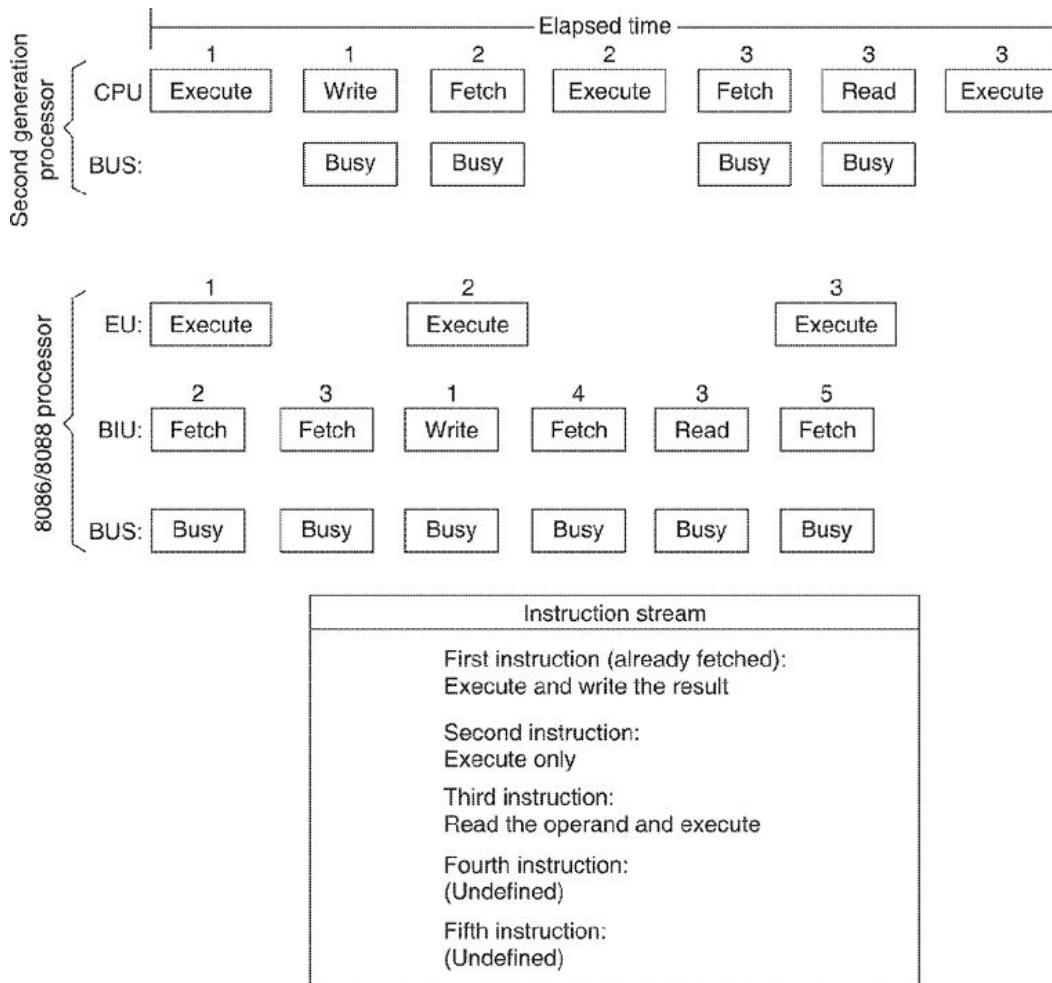


Figure 5.2 Overlapped instruction fetch and execution.

Register set

The 8086 CPU has fourteen 16-bit registers as shown in Figure 5.3. These registers are divided into Data Register Group (four registers), Segment Register Group (four registers) and Pointer and Index Register Group (four registers). The two remaining registers are Instruction Register (Program Counter) and Flag Register.

Data registers

The 8086 has four 16-bit data registers AX, BX, CX, and DX. These registers are unique since their upper and lower bytes can be addressed separately in addition to being single 16-bit registers. Thus, they may be treated as four 16-bit registers or eight 8-bit registers as shown in Figure 5.3. The functions of data registers are as follows:

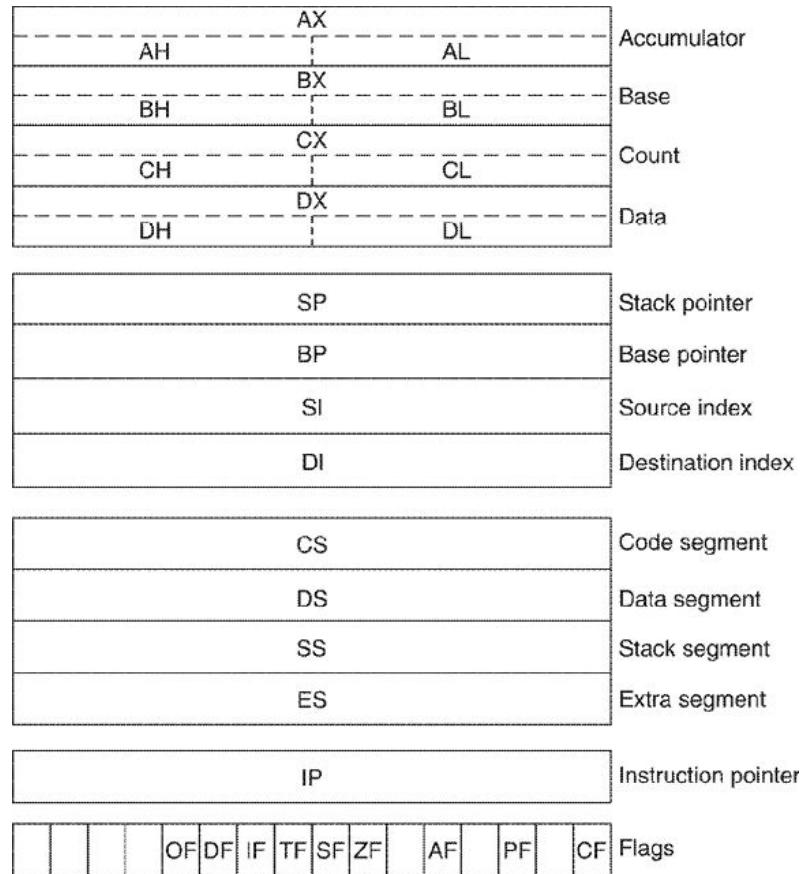


Figure 5.3 Intel 8086/8088 register set.

AX register: The AX register serves as a primary accumulator. It is unique in the following ways:

- Input/output operations pass data through AX (or AL).
 - Instructions involving AX (or AL) and immediate data usually require less program memory than that required by other registers.
 - Several powerful string primitive instructions require one of the operands to be AX (or AL).
 - AX contains the one-word operand and the result in 16-bit multiply and divide instructions, whereas AL is used for 8-bit operations. In 32-bit multiply and divide instructions, AX is used to hold the lower-order word operand.

BX register: In addition to serving as a general-purpose data register, BX can be used as base register while computing the data memory address.

CX register: In addition to serving as a general-purpose data register, it can be used to hold count in multi-iteration instructions. Several Intel 8086 instructions can be made to repeat or to loop. In such instructions, CX holds the desired number of repetitions and is automatically decremented after each iteration. When CX becomes zero, the execution of instructions is terminated. Similarly, the 8-bit CL register is used as count register in bit-shift and rotate instructions.

DX register: In addition to serving as a general-purpose data register, DX may be used in I/O instructions, multiply and divide instructions. DX contains the addresses of the I/O ports in certain types of I/O instructions. In 32-bit multiply and divide instructions, DX is used to hold the high-order word operand.

Segment registers

Intel introduced the concept of memory segmentation in the 8086. In memory segmentation, memory is divided into a number of parts called segments. Memory segmentation and its benefits have been discussed in Chapter 2. In the 8086, the 1 MB physical memory is divided into four segments—Code Segment, Data Segment, Stack Segment and Extra Segment. Each segment has memory space of 64 KB (Figure 5.4). Each segment is addressed by a 16-bit segment register as follows:

CS—Code Segment Register

DS—Data Segment Register

SS—Stack Segment Register

ES—Extra Segment Register

The 8086 memory address is 20 bits. The segment register supplies the higher-order 16 bits of the 20-bit memory address. All memory addresses of the 8086 are computed by summing the contents of the segment register (shifted left by 4 bits) and an offset address.

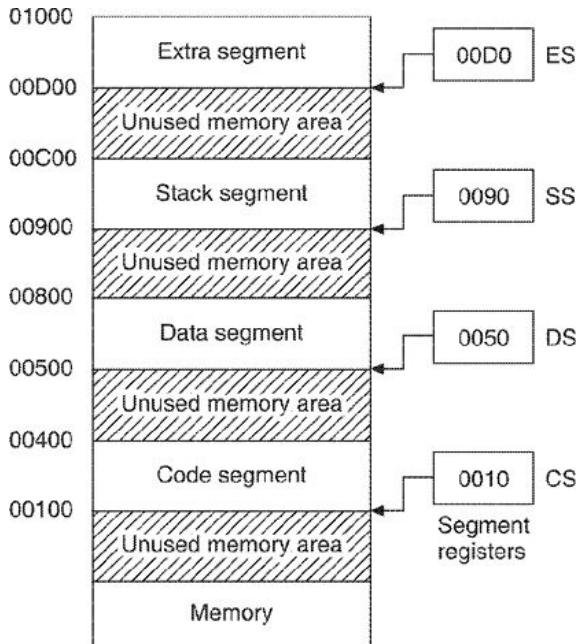


Figure 5.4 Memory segmentation.

The offset address is calculated in different ways for different addressing modes. The selected segment register contents are left shifted by 4 bits. This now represents the starting address of the segment in memory. The effective address, i.e. address basically represents the offset from the starting address of the segment. The offset address is added to segment register contents after 4 bit left-shift, to get the effective address, i.e. the actual memory location address. It may be represented as

Segment register contents	—	XXXX	(Hex)
Segment register contents	—	XXXX0	(Hex)
(After 4 bit left shift)			
<u>Offset address</u>	—	<u>YYYY</u>	(Hex)
Effective address	—	XZZZY	(Hex)

Figure 5.5 shows an example of memory segmentation.

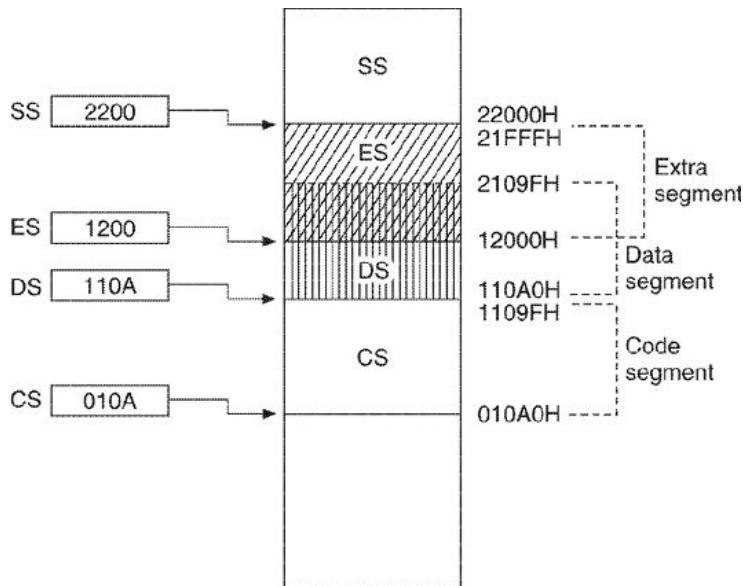


Figure 5.5 An example of memory segmentation.

The starting addresses of the various segments and the segment register contents are:

Segment	Starting address (in Hex)	Segment register contents (in Hex)
Code Segment	010A0	CS—010A
Data Segment	110A0	DS—110A
Extra Segment	12000	ES—1200
Stack Segment	22000	SS—2200

You will notice that Data Segment and Extra Segment have overlapped space which may be addressed through either of the segment registers. Thus, there may be two logical addresses for one location.

If a location 109F0 (Hex) of Code Segment is to be addressed to fetch an instruction, the physical address will be calculated as follows:

$$\text{Code Segment Register} = 010A \text{ (Hex)}$$

$$\text{Code Segment Register after 4 bit left-shift} = 010A0 \text{ (Hex)}$$

Offset Address = Offset within Code Segment = Contents of the Instruction

$$\text{Pointer} = F950 \text{ (Hex)}$$

$$\begin{array}{r} 010A0 \\ + F950 \end{array}$$

$$\begin{array}{l} \text{Effective address, i.e. address of the} \\ \text{actual memory location} \end{array} \quad \begin{array}{l} 109F0 \text{ (Hex)} \end{array}$$

You will notice that all the registers used for memory calculation are of 16 bit length and we are able to generate 20-bit address by using them.

Pointer and index registers

The registers in this group are as follows:

- Stack Pointer (SP)
- Base Pointer (BP)
- Source Index (SI)
- Destination Index (DI)
- Instruction Pointer (IP)

Stack Pointer (SP): The stack pointer is used in instructions which use stack, i.e. PUSH, POP, CALL, RET, etc. It always points to a location in memory known as the stack top. However, the complete address is formed by adding the contents of the stack segment register, after 4 bit left-shift, to the stack pointer as explained earlier.

Base Pointer (BP): The chief purpose of this register is to provide indirect access to data in stack register. It may also be used for general data storage.

Source index (SI) and Destination index (DI): These registers may be used for general data storage. However, the main purpose of these registers is to store offset in case of indexed, base indexed and relative base indexed addressing modes.

Instruction Pointer (IP): This register is also referred as Program Counter. As the name suggests, it is used for the calculation of actual memory addresses of instructions. It stores the offset for the instruction. During an instruction fetch, IP contents are added to the Code Segment register contents after 4 bit left-shift as explained earlier.

Flag register

The flag register is also referred as Program Status Word. Figure 5.6 shows the bit assignments of the flag register. It has 9 active flags out of which 6 are status flags and 3 are control flags.

The status flags are Carry, Parity, Auxiliary Carry, Zero, Sign and Overflow. These status flags reflect conditions produced by the execution of arithmetic or logic instructions.

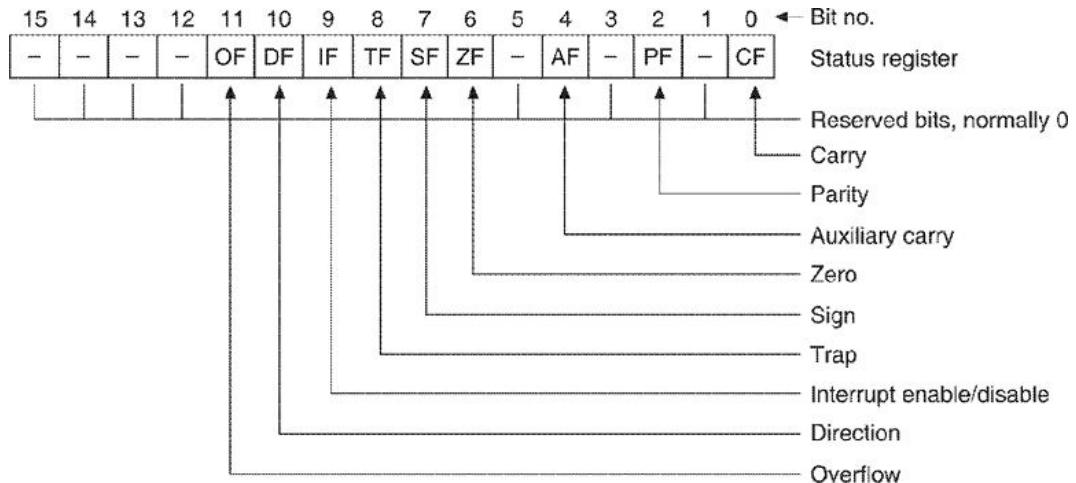


Figure 5.6 Intel 8086 flag register.

Carry Flag (CF): Carry is set after an arithmetic operation results in a carry out of MSB or a borrow in subtraction. This flag is also used in some shift and rotate instructions.

Parity Flag (PF): It is set if the result of byte operation or lower byte of the word operation contains an even number of 1s.

Auxiliary Carry Flag (AF): The flag is set if there is a carry out of the lower nibble to the higher nibble of an 8-bit quantity (or low-order byte of a 16-bit quantity). It is used for BCD operations.

Zero Flag (ZF): The zero flag is set whenever the result of the operation is zero.

Sign Flag (SF): The sign flag is set if, after the arithmetic or logic operations, the MSB of the result is 1. It indicates that the result is negative.

Overflow Flag (OF): This flag is used to detect magnitude overflow in signed arithmetic. For addition operation, the flag is set when there is a carry into the MSB and no carry out of the MSB or vice versa. For subtraction operation, the flag is set when the MSB needs a borrow and there is no borrow from MSB or vice versa.

There are three control flags—Direction Flag, Interrupt Enable Flag and Trap Flag. These can be set or cleared by program to alter processor operations.

Direction Flag (DF): It is used with string operations. When set, it causes the string instructions to auto decrement or to process strings from right to left. Otherwise the string is pursued in auto increment, i.e. from left to right.

Interrupt Enable Flag (IF): This flag enables the 8086 to recognize the external interrupt requests. When IF = 0, all maskable interrupts are disabled. It has no effect on either non-maskable interrupts or internally generated interrupts.

Trap Flag (TF): Setting the TF bit puts the processor into single step mode for debugging.

5.3 PIN DESCRIPTION

The pin configuration of the 8086 has been shown in Figure 5.7. As mentioned before, the 8086 can work either in minimum mode or in maximum mode by assigning the pin $\text{MN}/\overline{\text{MX}}$ to 5 V or GND. In minimum mode ($\text{MN}/\overline{\text{MX}} = 5 \text{ V}$), the 8086 works in single processor environment whereas in maximum mode ($\text{MN}/\overline{\text{MX}} = \text{GND}$), it works in multiprocessor complex environment. A set of the 8086 pins change their functions based on the mode set by the $\text{MN}/\overline{\text{MX}}$ pin. Other pins have common functions in both the modes.

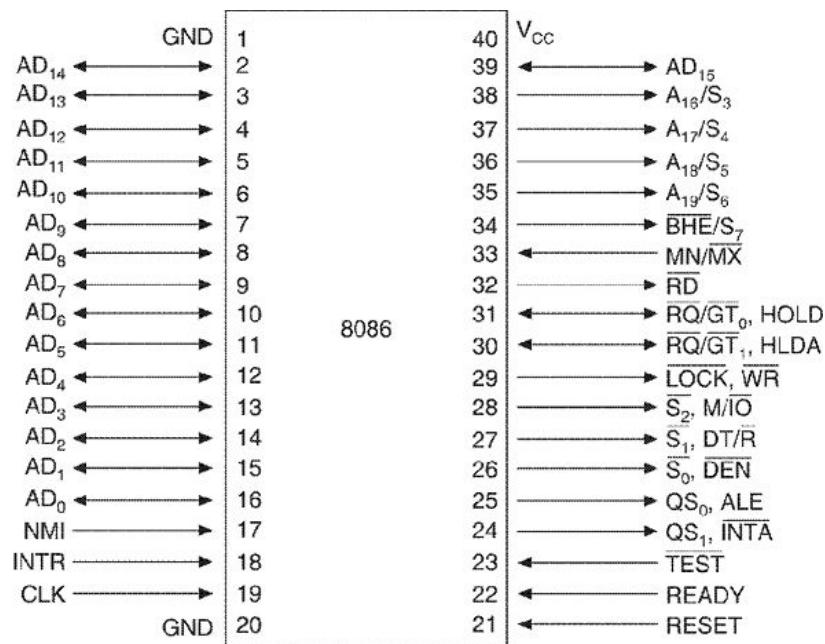


Figure 5.7 Intel 8086 pin diagram.

Pins with common functions

Pin no.	Name	Type	Description
39, 2– 16	AD ₁₅ – AD ₀	Bidirectional Tristate	The signals have dual function as in case of the 8085. They act as address bus during the first part of machine cycle and as data bus in the later part.
35– 38	A ₁₉ /S ₆ – A ₁₆ /S ₃	Output Tristate	Contains address information in the first part and status bits in the later part. The status bits, when decoded, indicate the type of operations (e.g. memory access) being performed and the segment register being used (see Figure 5.8).

34		Output Tristate	<p>\overline{BHE} is output during the first part of machine cycle. Low signal on \overline{BHE} pin indicates access to high-order memory bank governed by data bits AD₁₅–AD₈; otherwise, access is only to the low-order memory bank governed by data bits AD₇–AD₀. \overline{BHE} and A₀ decide the memory banks and types of access (see Figure 5.9). S₇ = 1 is output during the later part of machine cycle. No function has been assigned to S₇.</p>
32		Output Tristate	<p>Read Control Signal. \overline{RD} is low when the 8086 is receiving data from memory or I/O device.</p>
22	READY	Input	<p>Wait State Request Signal. A high READY input early in a machine cycle causes the 8086 to extend the machine cycle by inserting wait states.</p>
23		Input	<p>Used in conjunction with WAIT instruction. The instruction puts the 8086 in idle state which ends only when the \overline{TEST} input goes low.</p>
18	INTR	Input	<p>INTR is the level-triggered interrupt signal (same as in the 8085). Sampled during the last clock cycle of each instruction.</p>
17	NMI	Input	<p>NMI (Non Maskable Interrupt) is positive edge triggered non maskable interrupt request.</p>
19	CLK	Input	<p>CLK is single clock signal from external crystal controlled generator. The 8086 requires clock signal with 33% duty cycle. Following are clock frequencies for different versions of the 8086.</p> <p>8086—..... 5 MHz 8086-2 8 MHz 8086-1 10 MHz</p>
			<p>The 8284 clock generator chip of Intel is widely used for this purpose.</p>
21	RESET	Input	<p>System Reset signal must be high for at least 4 clock periods to cause reset. When the 8086 is reset, then:</p> <ul style="list-style-type: none">(i) The flag register is cleared. This disables the external interrupts.(ii) Instruction Pointers and three segment registers DS, SS and ES are cleared.(iii) The Code Segment register (CS) is set to FFFF (Hex). <p>Thus, immediately after reset, the program execution will start with the instruction stored at FFFF0H. Reset operations mentioned above take about 10 clock periods during which no operation takes place.</p>
40	V _{CC}		<p>Used to supply +5 V input supply voltage. The 8086 requires supply with $\pm 5\%$ tolerance.</p>
1, 20	GND		<p>For reliable operations, both pins 1 and 20 must be grounded.</p>
33		Input	<p>A high on this pin selects the minimum mode, whereas a low selects the maximum mode.</p>

S ₄	S ₃	Segment register
0	0	ES
0	1	SS
1	0	CS or no access
1	1	DS

S₅ = IF (Interrupt Enable Flag)

S₆ = 0 (Always)

Figure 5.8 Function of status bits S₆–S₃.

BHE	A ₀	Data access
0	0	Both high-order and low-order banks accessed for word read/write
0	1	High-order bank accessed from odd address for byte read/write
1	0	Low-order bank accessed from even address for byte read/write
1	1	None

Figure 5.9 $\overline{\text{BHE}}$ characteristics.

Minimum mode pin function assignments

Pin no.	Name	Type	Description
25	ALE	Output	Address Latch Enable. Since data and address are multiplexed on a single bus, ALE is output high to identify a valid address.
29		Output Tristate	Write Control Signal. $\overline{\text{WR}}$ is pulsed low by the 8086 when data is sent to memory or I/O through data bus.
28		Output Tristate	During the read write operation, signal on this pin identifies whether memory ($\text{M}/\overline{\text{IO}} = \text{high}$) or I/O port ($\text{M}/\overline{\text{IO}} = \text{low}$) is being accessed.
24		Output	The $\overline{\text{INTA}}$ is the acknowledgement of interrupt request on INTR pin. $\overline{\text{INTA}}$ is pulsed low in two consecutive bus cycles. The first pulse indicates interrupt acknowledgement. During the second pulse, external logic puts the ‘interrupt type’ on the data bus. It is received by the 8086 and ISR location is determined.
27		Output Tristate	Data Transmit/Receive. This signal, when high, indicates that data is being transmitted by the 8086. The low signal indicates that the 8086 is receiving the data. Thus, this signal is used to control data flow direction.
26		Output Tristate	Data Bus Enable. This signal, when low, indicates that the microprocessor address/data bus is to be used as data bus. It can be used to enable data bus buffers.
31	HOLD	Input	Hold request. This signal, when high, indicates that another master has requested for direct memory access. When HOLD becomes low, it indicates that direct memory access is no more required.
30	HLDA	Output	The microprocessor sends high signal on HLDA to indicate acknowledgement of DMA request. It then tristates the buses and control signals. When HOLD becomes low, the microprocessor makes HLDA low and regains the control of buses.

Maximum mode pin function assignments

The maximum mode is used for multiprocessor environment in which a number of processors share the bus.

Pin no.	Name	Type	Description
28, 27, 26		Output Tristate	These three signals are decoded by the bus controller to provide eight separate control signals, as shown in Figure 5.10.

24, 25	QS ₁ , QS ₀	Output	These two signals are decoded to provide instruction queue status as shown in Figure 5.11.
29		Output Tristate	This signal indicates that an instruction with lock prefix is being executed and the bus is not to be used by other processors.
30, 31	and	Bidirectional	In maximum mode, HOLD and HLDA signals are converted to bidirectional signals \overline{RQ} (Bus Request)/ \overline{GT} (Bus Grant). The operation is same as HOLD and HLDA. Out of $\overline{RQ}/\overline{GT}_0$ and $\overline{RQ}/\overline{GT}_1$, the $\overline{RQ}/\overline{GT}_0$ has higher priority.

\overline{S}_2	\overline{S}_1	\overline{S}_0	Control function
0	0	0	Interrupt acknowledge
0	0	1	I/O read
0	1	0	I/O write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	No operation

Figure 5.10 Decoding of signals.

QS ₁	QS ₀	Decoded function
0	0	Queue is an idle state
0	1	First byte of opcode has entered queue
1	0	Queue empty
1	1	Subsequent byte of opcode has entered queue

Figure 5.11 Decoding of QS₀, QS₁ signals.

Pins and signals—Intel 8088

Figure 5.12 shows the pin diagram of the 8088 chip. You would notice that majority of the pins are the same as in 8086. The major difference between the two processors is that 8088 has an 8-bit external data bus, although it is internally a 16-bit microprocessor. The following pins and signals in the 8088 are different from those in the 8086.

Pin no.	Name	Type	Description
9– 16	AD ₇ – AD ₀	Bidirectional Tristate	Multiplexed Address/Data bus.
39, 2– 8	A ₁₅ – A ₈	Output Tristate	Higher Address Bus
28		Output Tristate	Memory/IO Port selection signal. IO/M = High indicates that I/O port is being addressed, IO/M = Low indicates that memory is being addressed. This is same as the 8085.

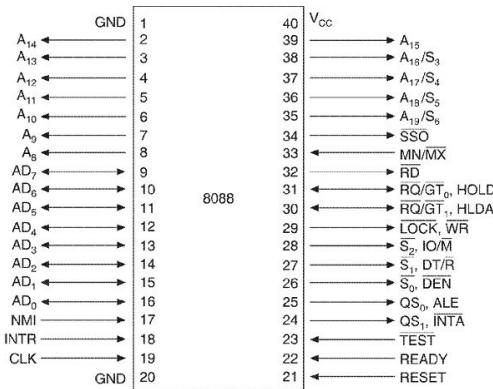


Figure 5.12 Intel 8088 pin diagram.

$\overline{IO/M}$	$\overline{DT/R}$	\overline{SSO}	Function
0	0	0	Interrupt acknowledge
0	0	1	Memory read
0	1	0	Memory write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	I/O read
1	1	0	I/O write
1	1	1	Passive (No Operation)

Figure 5.13 Decoded status bits for Intel 8088 bus cycles.

5.4 EXTERNAL MEMORY ADDRESSING

The data stored in the 8086 memory may be 8-bit bytes or 16-bit words. They can occupy different memory locations without any restrictions. To facilitate memory read/write operations in which either byte or word data may be read/written, the 8086 memory is organized in byte form.

The memory is organized into two banks (Figure 5.14):

- Low-order memory having even addresses like 00000, 00002, 00004, etc.
- High-order memory having odd addresses like 00001, 00003, 00005, etc.

Clearly, an 8-bit data will occupy only one memory bank and reading or writing will require access to one bank only. The low-order or the high-order bank will depend on the address being even or odd. On the

other hand, a 16-bit data will have two bytes, of which the one with even address will occupy the low-order address bank, while the other with odd address will occupy the high-order address bank. For all addresses which are even, i.e. 0000, 0002, etc. the address bit A_0 is always zero. Thus, it is easy to identify and select the low-order memory bank for access based on $A_0 = 0$. For high-order memory bank access, the 8086 outputs a signal \overline{BHE} (Bank High Enable) which goes low whenever an odd address (0001, 0003, etc.) is output on the address bus. Thus it is possible to select different memory locations and read/write both bytes/words.

The Address Decoders for memory banks in Figure 5.14 get selected by A_0 or \overline{BHE} signal. They further select memory chip depending on the address range. Now, we shall illustrate through examples how the read/write operations take place.

- (a) 8-bit read at even address, e.g. 0A2F8H.
 - (i) Address information 0A2F8H is output on-/ the address bus.
 - (ii) Since $A_0 = 0$, a low-order bank will be selected. \overline{BHE} remains high.
 - (iii) Memory chip will be selected by address decoder based on address range.
 - (iv) Data from 0A2F8H location will be output by memory on the data bus AD₀-AD₇.

The operation is shown in Figure 5.15.

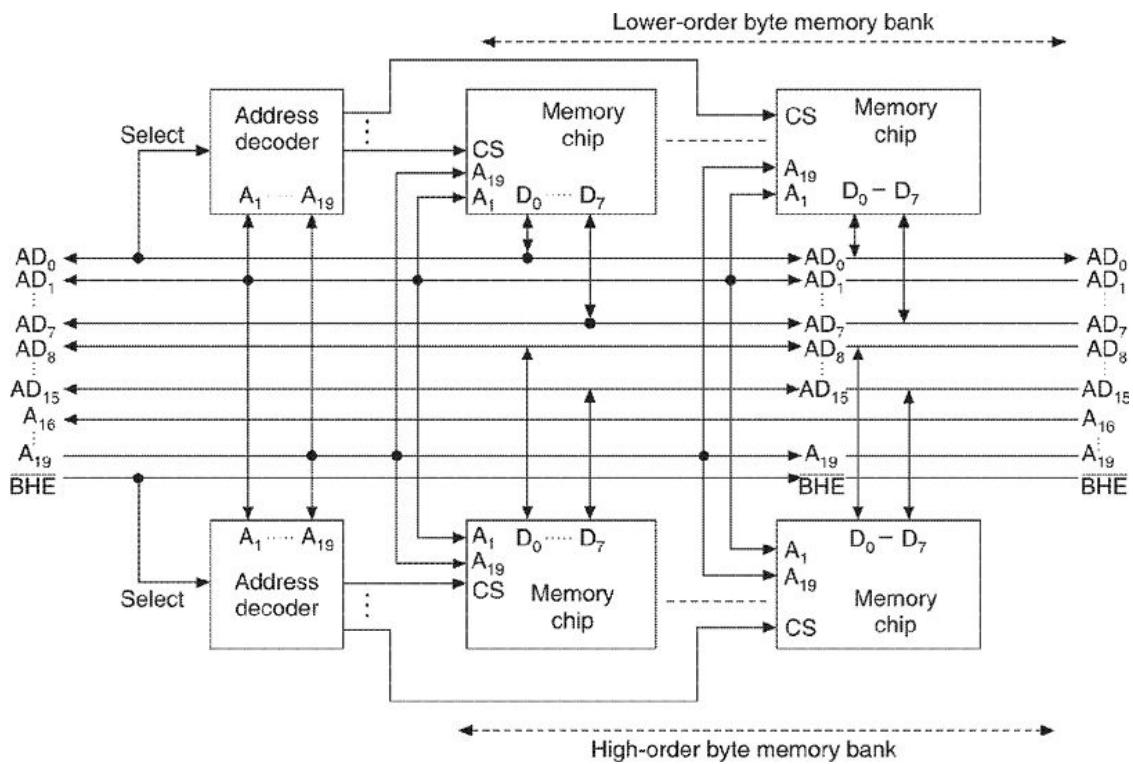


Figure 5.14 External memory addressing.

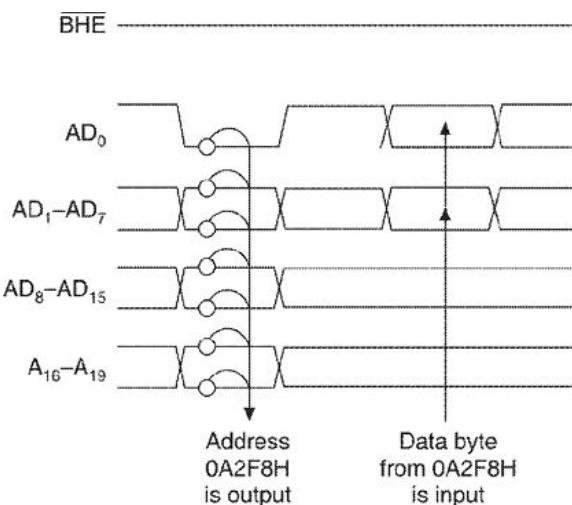


Figure 5.15 Single byte read at even address.

(b) 8-bit read at odd address, e.g. 0A2F7H.

- (i) $\overline{\text{BHE}}$ is pulsed low as the address 0A2F7H is output on the address bus.
- (ii) Memory chip in the high-order bank (since $\overline{\text{BHE}} = \text{Low}$) is selected by the address decoder based on the address range.
- (iii) Data from 0A2F7H is output by memory on data bus AD₈-AD₁₅.
- (iv) The 8086 will place the data in register specified in the instruction, e.g. if AL is specified the data will go to lower byte

of AX and if AH is specified it will go to higher byte of AX. This is done by the 8086 automatically.

The operation is shown in Figure 5.16.

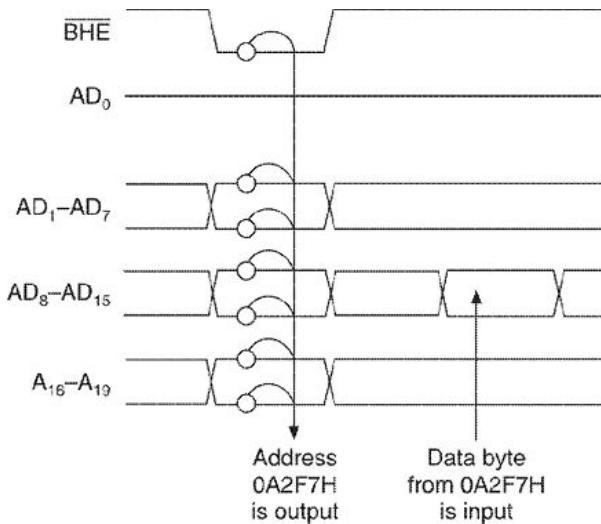
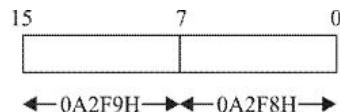


Figure 5.16 Single byte read at odd address.

- (c) 16-bit read from even byte address boundary, i.e. lower byte at even address, e.g. 0A2F8H, 0A2F9H locations.



- (i) \overline{BHE} is pulsed low for 16-bit memory read/write in the first part of the bus cycle. In this case, since A = 0, both low and high order banks will be selected.
- (ii) Address information 0A2F8H is output on address bus. Note that the address is decoded from address bits A₁ to A₁₉, (A₀ is used only for selection of memory bank). Address decoders (in both memory banks) will select the memory chips based on the address range. Both the memory locations 0A2F8H and 0A2F9H are enabled.
- (iii) Data from 0A2F8H is put on the data bus AD₀-AD₇ and data from 0A2F9H is put on the data bus AD₈-AD₁₅.
- (iv) The 8086 stores the data in the specified register.

The operation is illustrated in Figure 5.17.

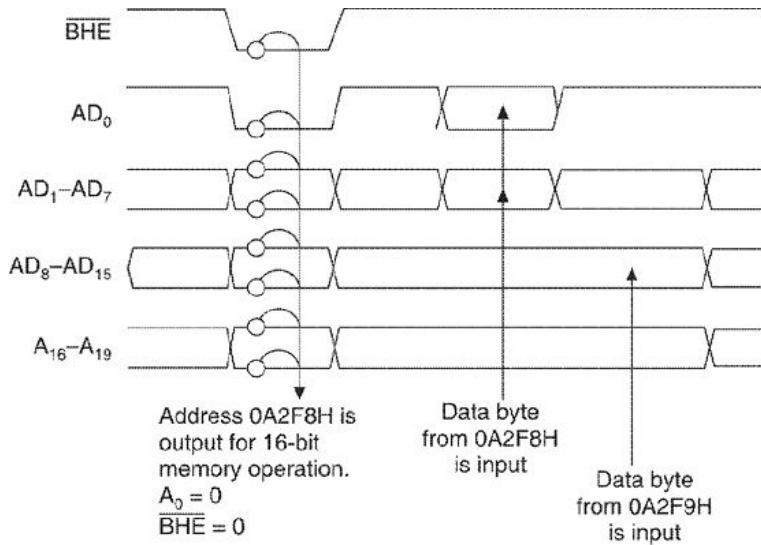
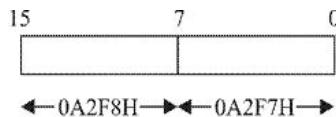


Figure 5.17 16-bit memory read from even address boundary.

- (d) 16-bit read from odd address boundary, i.e. lower byte at odd address, e.g. 0A2F7H, 0A2F8H locations.



- (i) In this case, $0A2F7H$ is output on the address bus. \overline{BHE} is pulsed low and since $A_0 = 1$, only the high-order memory bank is selected. The address decoder selects the memory chip based on the address range.
- (ii) Data from memory address $0A2F7H$ is put on the data bus AD_8-AD_{15} .
- (iii) Memory Address $0A2F8H$ is put on the address bus. $A_0 = 0$ selects the low-order memory bank. The address decoder selects the memory chip based on the address range.
- (iv) Data from memory address $0A2F8H$ is put on the data bus AD_0-AD_7 .
- (v) The 8086 automatically places data from AD_0-AD_7 in higher byte and from AD_8-AD_{15} in lower byte of the word.

Figure 5.18 illustrates the operation.

Thus it is clear that 16-bit memory operation with even address boundary takes less time than that with odd address boundary. It is, therefore, advisable to place word arrays at even address boundary. The 8086 Assembler has ‘EVEN’ directives for this purpose.

In case of the 8088, \overline{BHE} signal is not present. Thus, memory is not organized as high and low memory banks but as a byte-oriented memory.

The memory operations are similar to 8085 or 8-bit read as explained above without bank selection.

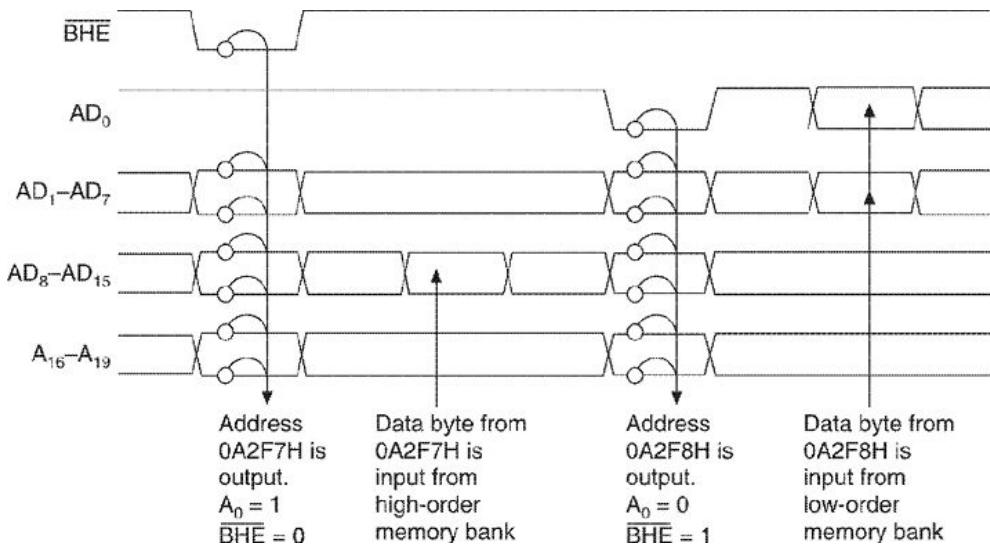


Figure 5.18 16-bit memory read from odd address boundary.

5.5 BUS CYCLES

We must remember that EU and BIU work asynchronously. The EU takes instructions from the instruction queue and executes them continuously. The only time EU waits for BIU to handover the instruction is when the program starts or when the program executes a branch instruction. The only other instant EU waits for BIU is when BIU is executing data memory access for EU.

Therefore, the instruction fetch cycle (as in case of the 8085) has been virtually eliminated in the 8086. Also, the bus cycles are BIU phenomenon. We shall now discuss memory and I/O read/write bus cycles for both minimum and maximum modes.

5.5.1 Memory or I/O Read for Minimum Mode

In minimum mode, MN/MX is placed at 5 V level. The timing for memory read bus cycle is shown in Figure 5.19.

The following activities take place during different clock cycles:

T₁

- ALE is pulsed high.
- BHE is made high/low depending on 8/16 bit read at odd/even address boundary.

- M/\bar{IO} is made high to indicate memory operation. It remains high during the entire bus cycle.
- DT/\bar{R} is low and remains low throughout the cycle, to indicate the direction of data transfer as memory to the processor.
- Address is put in the address bus. The falling edge of ALE is used to latch the address from the address bus.

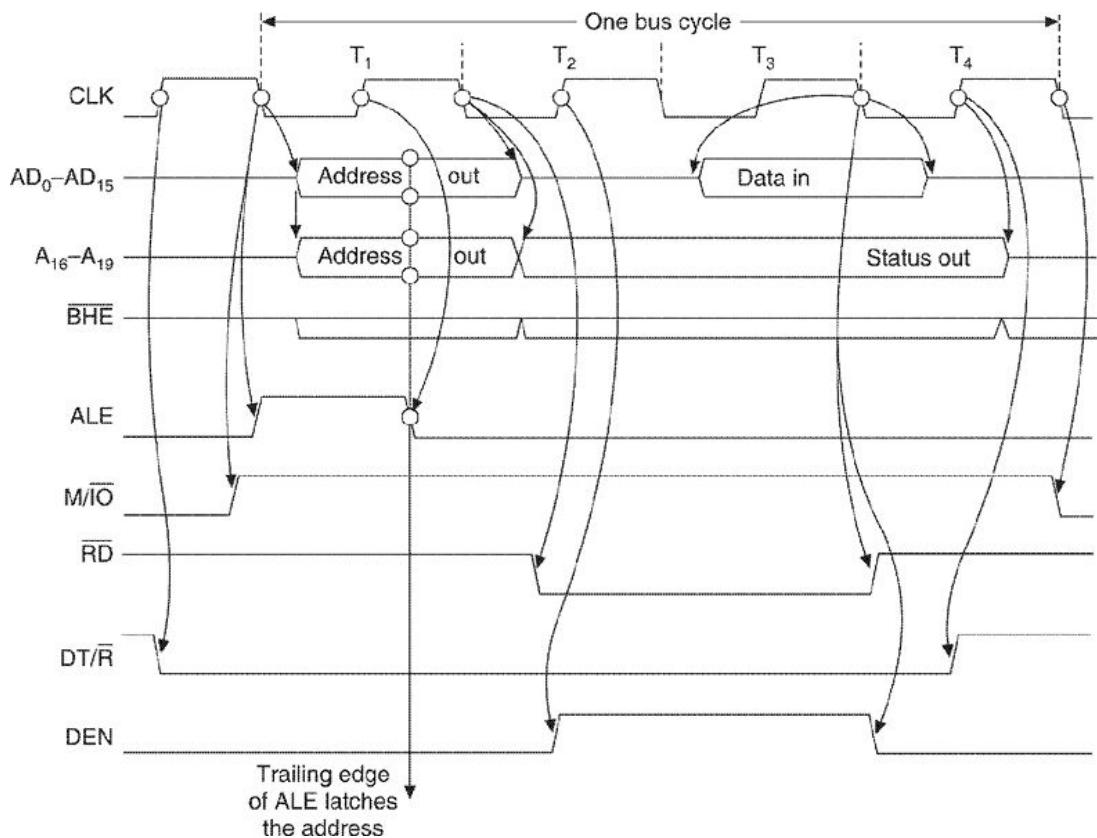


Figure 5.1 Memory read bus cycle in minimum mode.

T₂

- Bus is turned around.
- \bar{RD} goes low as read-control signal.
- DEN goes high to enable the 8286 transceiver.
- \bar{BHE} goes high if it was made low in T₁.
- Status is put on the A₁₆-A₁₉ lines. The activity starts in T₂ and continues till T₄.

T₃

- DEN goes low.
- Data is put on lines AD₀–AD₁₅.

T₄

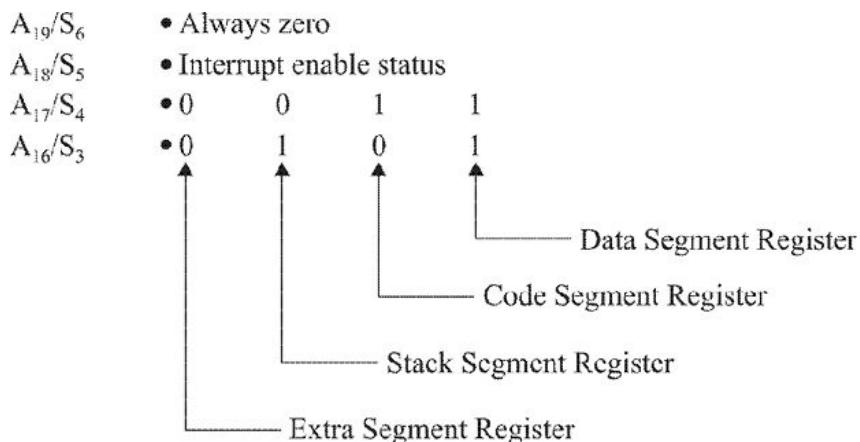
- $\overline{M/IO}$ goes low.
- \overline{RD} goes high.

The I/O read bus cycle will have only one signal that is different, i.e. $\overline{M/IO}$ will go low in T₁ and will become high in T₄.

In case of the 8088 memory read cycle, $\overline{M/IO}$ is replaced by $\overline{IO/M}$ which will go low in T₁ to signify memory operation. \overline{BHE} is absent, and the address/data bus AD₀–AD₁₅ is replaced by the address/data bus AD₀–AD₇ and address bus A₈–A₁₅. The rest of the signal timings are almost same.

For I/O read, $\overline{IO/M}$ will go high in T₁ and will remain high throughout. Other signals will remain same.

The status bits output on A₁₆–A₁₉ during T₂ will be as per the following:



5.5.2 Memory or I/O Write for Minimum Mode

The timing of memory write bus cycle is illustrated in Figure 5.20. The following are the activities that take place during the various clock cycles.

T₁

- ALE is pulsed high.

- $\overline{\text{BHE}}$ is made high/low depending on 8/16 bit write at odd or even address boundary.
- $\overline{\text{M/IO}}$ is made high to indicate memory operation. It remains high during the entire bus cycle.
- $\overline{\text{DT/R}}$ is high and remains high throughout the cycle to indicate the direction of proposed data transfer from processor to memory.
- Address is put on the address bus. The falling edge of ALE is used to latch the address from the address bus.
- DEN goes high to enable the 8286 transceivers.

T₂

- $\overline{\text{WR}}$ goes low as write control signal.
- $\overline{\text{BHE}}$ goes high if it was made low in T₁.
- Bus is turned around.
- Status is put on the A₁₆–A₁₉ lines. The activity starts during T₂ and continue till T₄.

T₃

- Data is put on the AD₀–AD₁₆ lines.

T₄

- $\overline{\text{M/IO}}$ goes low.
- $\overline{\text{WR}}$ goes high.
- $\overline{\text{DT/R}}$ goes low.
- DEN goes low.

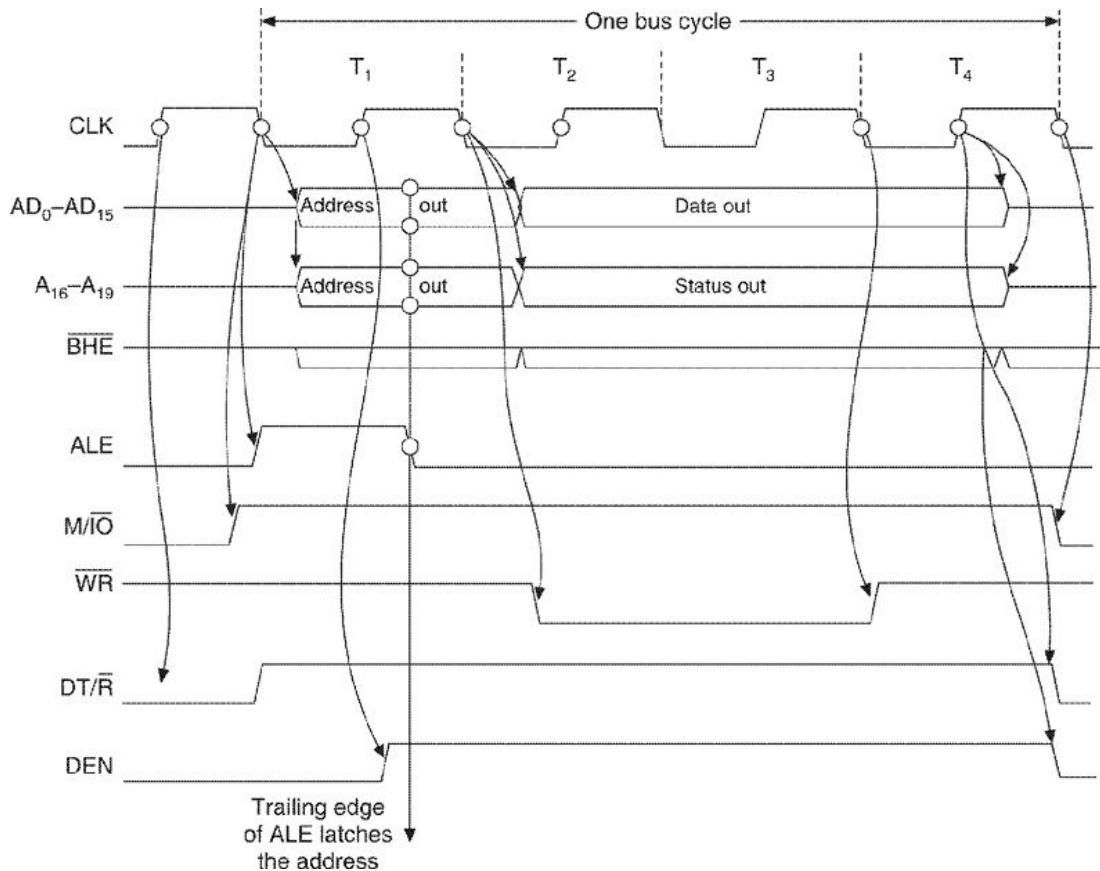


Figure 5.20 Memory write bus cycle in minimum mode.

You must have noticed that the logic for both read/write operations is the same. The only difference being the use of \overline{WR} in place of \overline{RD} . The status bit configuration is the same as explained earlier in the read cycle.

The I/O write cycle has M/\overline{IO} signal as low instead of high. The rest of the signal activities remain the same.

In case of the 8088 memory write cycle, M/\overline{IO} is replaced by $\overline{IO/M}$ which will go low in T₁ to signify memory operation. \overline{BHE} is absent and the address/data bus AD₀-AD₁₅ is replaced by the address/data bus AD₀-AD₇ and the address bus A₈ to A₁₅. The rest of the signal timings are almost the same.

For I/O write, $\overline{IO/M}$ will go high in T₁ and will remain high throughout. Other signals will be the same.

5.6 SOME IMPORTANT COMPANION CHIPS

We shall now describe the following chips which are used for the design of the 8086-based systems.

- Clock Generator Intel 8284

- Bidirectional Bus Transceiver Intel 8286/8287
- 8 Bit Input/Output Port Intel 8282/8283
- Bus Controller Intel 8288

The first three chips have been used extensively, in both minimum and maximum mode systems. The Bus Controller 8288 is used mainly in maximum mode system. We shall present only the essential features of these chips here. The designers may consult data books for other details.

5.6.1 Clock Generator Intel 8284A

Every 8086 microprocessor-based system has the clock generator 8284 as one of the components. The clock generator 8284 (an 18-pin IC) provides the following functions:

- A stable clock to the 8086 microprocessor as per specification (33% duty cycle).
- Facility to synchronize clock signals of other 8086 microprocessors in case of the multiprocessor environment.
- Reset signal in synchronization with clock as required by the 8086.
- Wait state logic.

The internal logic of the 8284A clock generator is shown in Figure 5.21 and the pin diagram of the chip is shown in Figure 5.22. In the following, we shall describe the functions of these pins and signals.

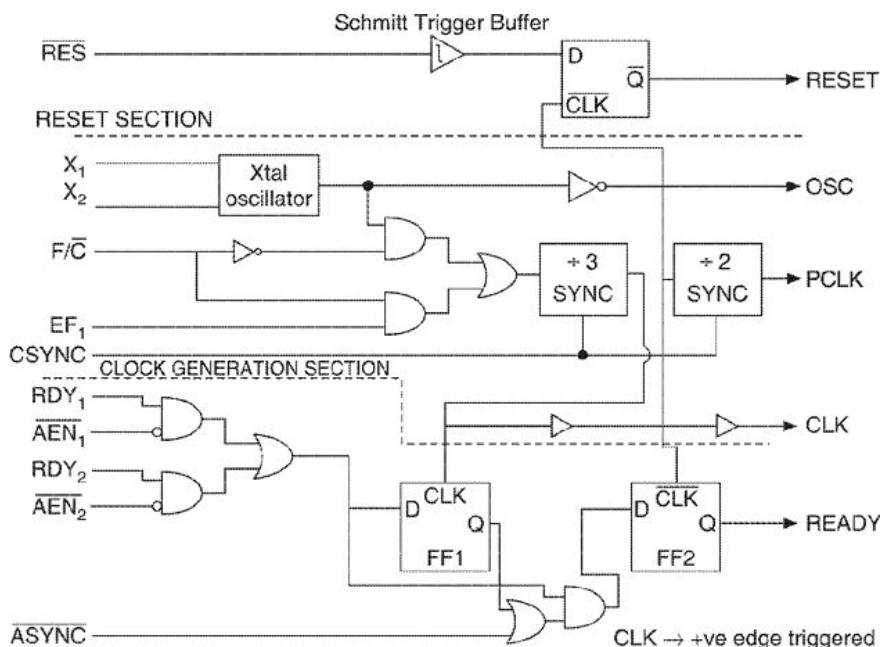


Figure 5.21 Internal logic of the clock generator Intel 8284.

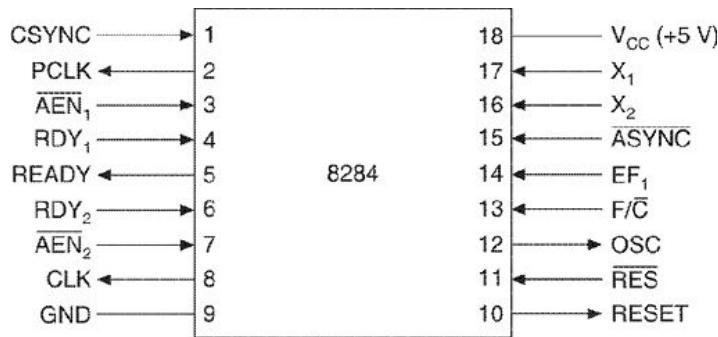


Figure 5.22 Clock generator Intel 8284—pin diagram.

Pin no.	Name	Type	Description
17, 16	X ₁ , X ₂	Input	External crystal connections. The clock frequency of the crystal connected between X ₁ and X ₂ must be exactly three times the required frequency. Thus, to achieve a clock frequency of 5 MHz (200 nanosecond clock period) the crystal frequency must be 15 MHz.
14	EF ₁	Input	Alternate clock input. It is possible to supply the externally generated clock signal at F ₁ . The clock signal at EF ₁ must be at the fundamental frequency.
13		Input	Clock source selection. The voltage on this pin determines whether the 8284 will take clock source from X ₁ , X ₂ or EF ₁ . If the input is high, clock at EF ₁ is taken; if low, the crystal connected at X ₁ , X ₂ is taken for clock frequency.
8	CLK	Output	CLK is MOS level signal designed to meet the requirements of the 8086. The pin is directly connected to the 8086/8088 CLK pin. The output signal has 33% duty cycle as required by the 8086.
12	OSC	Output	OSC is an oscillator output running at crystal or EF ₁ frequency. It can be used to provide clock signal at EF ₁ to the other 8284 clock generators in a multiprocessor system.
2	PCLK	Output	It is the TTL level clock signal output for support circuits. PCLK runs at half the frequency of CLK and has 50% duty cycle.
			Figure 5.23 illustrates the timing signal of crystal frequency or EF ₁ , and shows outputs at OSC, CLK and PCLK pins.
1	CSYNC	Input	This pin is used for synchronization of clock signals in multiprocessor environment where all processors receive clock at EF ₁ . When CSYNC is high, the 8284A clock generator/driver is stopped. When CSYNC goes low subsequently, the clock outputs restart. If the same CSYNC signal is input to a number of the 8284 devices which receive the same EF ₁ , all microprocessors will be exactly synchronized.
11		Input	Reset logic input. An external device can send a low signal to reset the 8086. Normally, this pin is connected to an R-C network for generating reset at power-on.
10	RESET	Output	Reset control signal output to Intel 8086. The 8086 requires that RESET signal must be synchronized with clock. The 8284, on receiving the RES signal, generates the synchronized RESET output as required by the 8086.
4, 6	RDY ₁ ,	Input	Wait state ready inputs. The slower memory or I/O devices can request for

	RDY ₂	extension of bus cycles using the RDY ₁ or RDY ₂ pin. Two wait state ready inputs have been provided to support multibus configurations.
5	READY	Output The READY input in the 8086 enables the bus cycle extension through wait state clock period insertion between T ₃ and T ₄ clock periods. The 8086 READY input must be synchronized with clock signal as shown in Figure 5.24. Based on one of the two inputs RDY ₁ or RDY ₂ , the 8284 clock generator produces an appropriately synchronized READY signal to the 8086.
3, 7	-	Input Two ready inputs RDY ₁ , RDY ₂ have been provided in the 8284 to support multibus configurations (Figure 5.25). A single Intel 8086 may be connected to two separate system buses, on which the data transfer may take place. The memory or I/O devices in any of the system buses may like to insert wait states. Thus, each system bus may have its own ready line. \overline{AEN}_1 and \overline{AEN}_2 have been provided to arbitrate bus priorities when both RDY ₁ and RDY ₂ are active. The 8284 responds to RDY ₁ only when \overline{AEN}_1 is low. Similarly, RDY ₂ is considered when \overline{AEN}_2 is low.
15	-	Input Ready synchronization selection input. It selects either one or two stages of synchronization for RDY ₁ and RDY ₂ inputs. If low, one level is selected; if high, two levels of synchronization are selected.
18	V _{CC}	- Connects to power source +5 V ± 10%.
9	GND	- Connects to system ground.

Figure 5.26 shows the simple interconnection between the 8086 microprocessor and the 8284 clock generator.

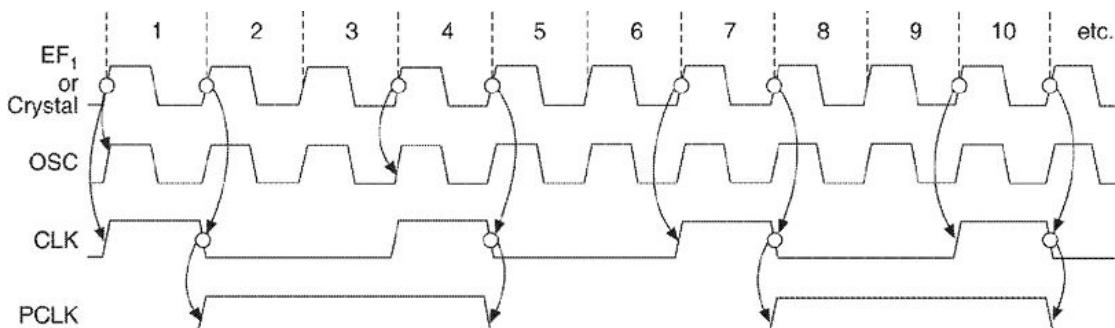
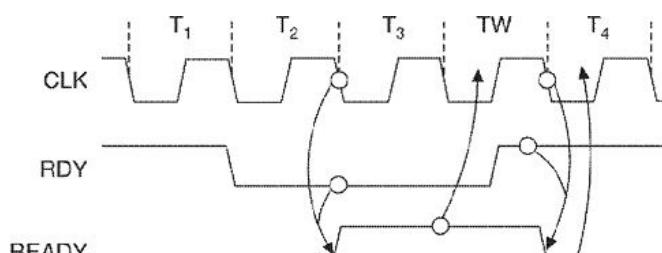


Figure 5.23 Clock generation at various pins.



RDY is input by external logic to the 8284 clock
READY is output by the 8284 clock, to be input to the 8086

Figure 5.24 READ signal synchronization.

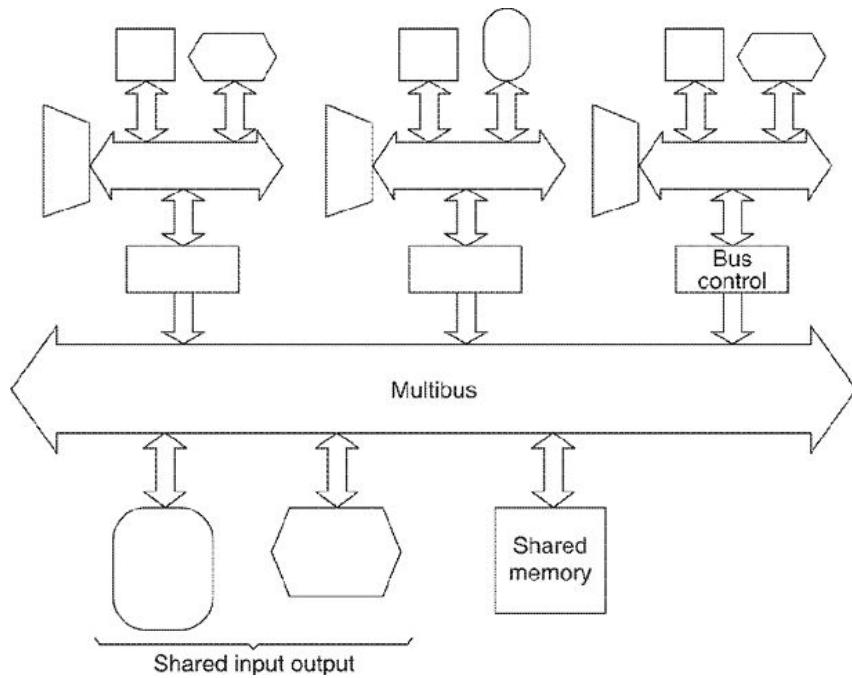


Figure 5.25 Intel multi-bus system structure.

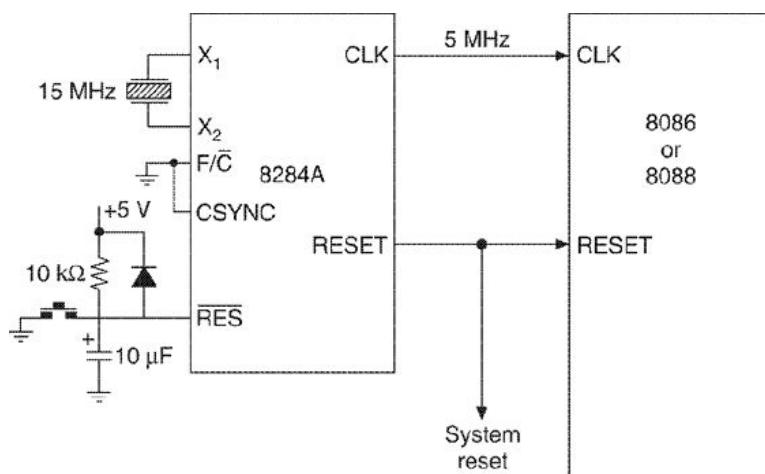


Figure 5.26 Interconnection between the 8086 and the 8284 clock generator.

5.6.2 Bidirectional Bus Transceiver Intel 8286/8287

The 8286 and the 8287 are basically bidirectional system bus buffers-cum-drivers. The pin diagram is shown in Figure 5.27. The pin description is as follows:

Pin no.	Name	Type	Description
1 to 8	A ₀ to A ₇	Bidirectional Tristate	Local bus. These connect to microprocessor data/address bus.
12 to 19	B ₇ to B ₀	Bidirectional Tristate	These lines connect to system bus of system.
9	-	Input	Output Enable

11	T	Input	Direction Select
10	GND	—	System Ground
20	V _{CC}	—	5 V Power Input

When T input is low, the data at B pins is output via the A pins. When T is high, the data at A pins is output via B pins. \overline{OE} must be held low for actual data transfer to take place. There is no difference between the two buses except that the system bus lines have a higher drive capacity.

The 8286 transfers data unaltered while the 8287 inverts the data at the time of the transfer.

The 8286 is widely used in the 8086-based systems as data bus transceivers. You would find T pin connected to DT/R, and \overline{OE} connected to \overline{DEN} .

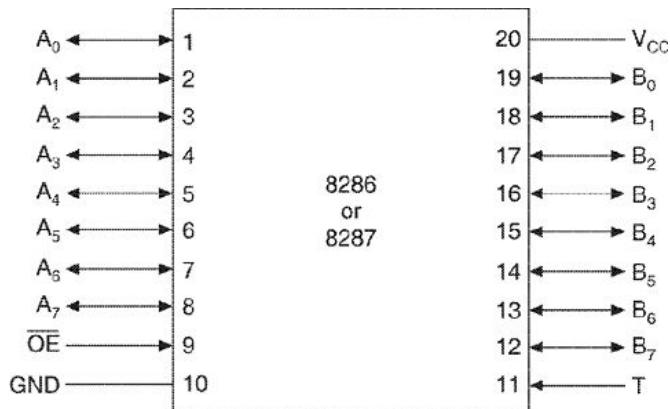


Figure 5.27 Bidirectional bus transceiver Intel 8286/8287 pin diagram.

5.6.3 8 Bit Input–Output Port Intel 8282/8283

The 8282 and 8283 are simply unidirectional latch buffers, like 74LS373. The difference between the 8282 and the 8283 is that while the 8282 does not alter the data, the 8283 inverts the input data. The pin diagram is shown in Figure 5.28. The description of pins functionalities is given below.

Pin no.	Name	Type	Description
1 to 8	DI ₀ to DI ₇	Input	Data input
19 to 12	DO ₀ to DO ₇	Output	Data output
9	—	Input	Output enable
11	STB	Input	Input data strobe
10	GND	—	System ground
20	V _{CC}	—	+5 V input power

When STB is high, the data on output pins track the data on input pins. On high-to-low transition of STB, the data is latched. The data remains unchanged when STB is low. The data is latched internally till \overline{OE} is low. When \overline{OE} is low the data is put on output lines. The 8282 outputs the data unaltered, whereas the 8283 inverts the data.

There is similarity of function and operation between the 8282 and 74LS373. In fact, 74LS373 is preferred by many designers.

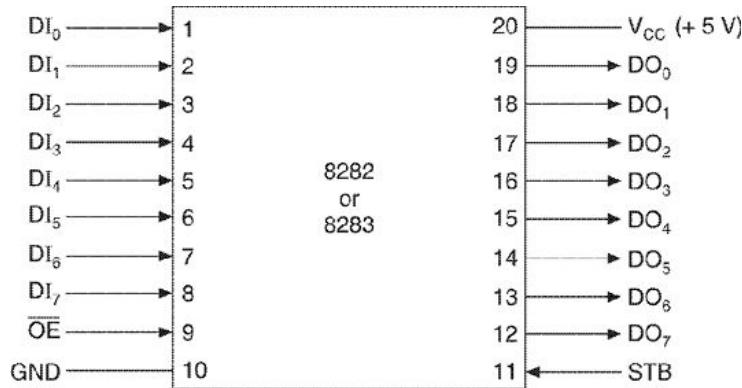


Figure 5.28 The 8282/8283 pin diagram.

5.6.4 Bus Controller Intel 8288

The bus controller Intel 8288 is used in maximum mode configuration of the 8086. Its main function is to decode the status line signals $\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$ and to generate system bus control signals. The 8288 can also be used to connect more than one Intel 8086 on one system bus, or to create more than one system bus. The pin diagram of the 8288 is shown in Figure 5.29. In the following, we describe the functions of different pins.

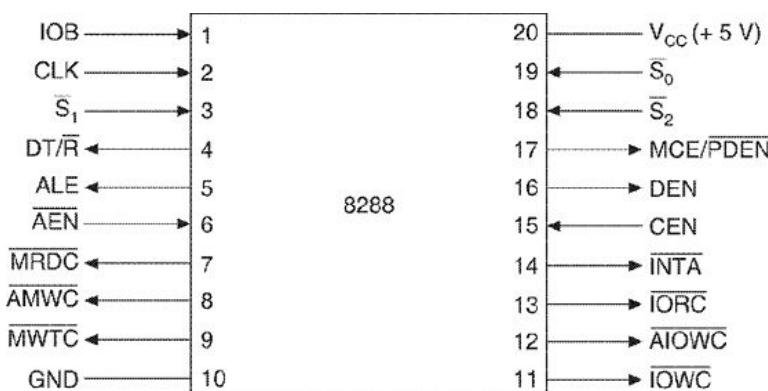


Figure 5.29 Bus controller Intel 8288—pin diagram.

Pin no.	Name	Type	Description
19, 3, 18		Input	Bus cycle status signals. These are decoded and control signals are generated.

2	CLK	Input	TTL clock signal. It is connected to CLK output from clock generator Intel 8284.
6, 15, 1	, CEN, IOB	Input	Bus priority and mode control signals. \overline{AEN} (Bus priority control/enable), CEN (Command Enable) and IOB (Mode Control) are used to control the generation of various control signals. The logic is described later.
7		Output Tristate	Memory Read Control Signal.
9		Output Tristate	Memory Write Control Signal.
8		Output Tristate	Advance Memory Write Control Signal. It has the same function as \overline{MWTC} , but is activated one clock pulse earlier.
13		Output Tristate	I/O Device Read Control Signal.
11		Output Tristate	I/O Device Write Control Signal.
12		Output Tristate	Advance I/O Device Write Control Signal. It is activate one clock pulse earlier than \overline{IOWC} ; otherwise it has the same function as \overline{IOWC} .
14		Output Tristate	Interrupt Acknowledge. It is output during two interrupt acknowledge bus cycles and is used as memory read control signal.
17		Output	Cascade/Peripheral Data Enable. It is used in configurations using Priority Interrupt Controller Intel 8259A.
5	ALE	Output	Address Latch Enable Signal.
4		Output	Data Direction Control Signal.
16	DEN	Output	Data Buffer Control Signal.
20	VCC		+5 V, Power Source
10	GND		Ground connected to system ground.

Memory and I/O devices connected to bus may use either \overline{IOWC} and \overline{MWTC} or \overline{AIOWC} and \overline{AMWC} as write control signal but not all four signals.

The 8086 and the 8088 control signals are same. The 8288 has two modes of operation—I/O Bus Mode and System Bus Mode. These modes can be selected by mode control pin IOB. When IOB = high, the 8288 bus controller goes to I/O Bus Mode and when IOB = low, it operates under System Bus Mode.

Under I/O Bus Mode (IOB = 1), signals \overline{MRDC} , \overline{MWTC} and \overline{AMWC} are floated all the time whereas signals \overline{IORC} , \overline{IOWC} , \overline{AIOWC} and \overline{INTA} are output and cannot be floated. Thus under I/O Bus Mode, the 8288 generates an I/O bus which is local I/O bus and cannot be shared with other microprocessors. Also, it cannot be used for DMA operation.

In the I/O Bus Mode, two control signals \overline{PDEN} and $\overline{DT/R}$ are used to drive I/O ports and line drivers. \overline{PDEN} is equivalent to \overline{DEN} , data enable

signal of the 8086, whereas $\overline{DT/R}$ is equivalent to DT/\overline{R} signal of the 8086 and is used for controlling bidirectional bus control.

Under System Bus Mode ($IOB = 0$), all the control signals (\overline{MRDC} , \overline{MWTC} , \overline{AMWC} , \overline{INTA} , \overline{IORC} , \overline{IOWC} and \overline{AIOWC}) are active. Thus normal system bus is generated. \overline{AEN} is used as bus enable control signal. \overline{AEN} is active only when $IOB = 0$, i.e. when the system bus is generated by the 8288.

IOB		Operation
0	0	Control signals are connected to system bus.
0	1	Control signals are floated.

$\overline{DT/R}$ can be used for direct memory access or priority arbitration.

DEN and $\overline{DT/R}$ are two standard buffer control signals generated in System Bus Mode. These two signals are the same as in the 8086. Where DEN is used as latching strobe, $\overline{DT/R}$ is used for direction control for bidirectional buffers.

The CEN (Command Enable) signal is used to disable (but not float) control signals. It can be used in both I/O Bus and System Bus Mode. When $CEN = 0$, control signals are inactive; otherwise when $CEN = 1$, control signals are active. It must be remembered that CEN does not float the signals, it just disables the logic.

Figure 5.30 shows signals \overline{AEN} , IOB , CEN and their effect on control signals.

The \overline{INTA} signal generated by the 8288 has the same function as in the 8086, as described earlier. The MCE control signal is used in large Intel 8086 microprocessor-based systems which use cascaded Intel 8259A priority Interrupt Control units.

Control input				Effect on control output			
IOB	CEN			$\overline{INTA}, \overline{IORC}, \overline{IOWC}, \overline{AIOWC}$		$\overline{MRDC}, \overline{MWTC}, \overline{AMWC}$	
0	0	0	System	Floated	Active	System	Floated
0	0	1	System	Floated	Inactive	System	Floated
0	1	0	System	Connected	Active	System	Connected
0	1	1	System	Connected	Inactive	System	Connected
1	0	0	I/O	Floated	Active	Not Used	Floated
1	0	1	I/O	Floated	Active	Not Used	Floated
1	1	0	I/O	Connected	Active	Not Used	Floated
1	1	1	I/O	Connected	Active	Not Used	Floated

Figure 5.30 Effect of \overline{AEN} , IOB and CEN on control signals.

5.7 MAXIMUM MODE BUS CYCLE

5.7.1 Memory Read Bus Cycle

The maximum mode memory read bus cycle is shown in Figure 5.31. The following activities take place in each clock period.

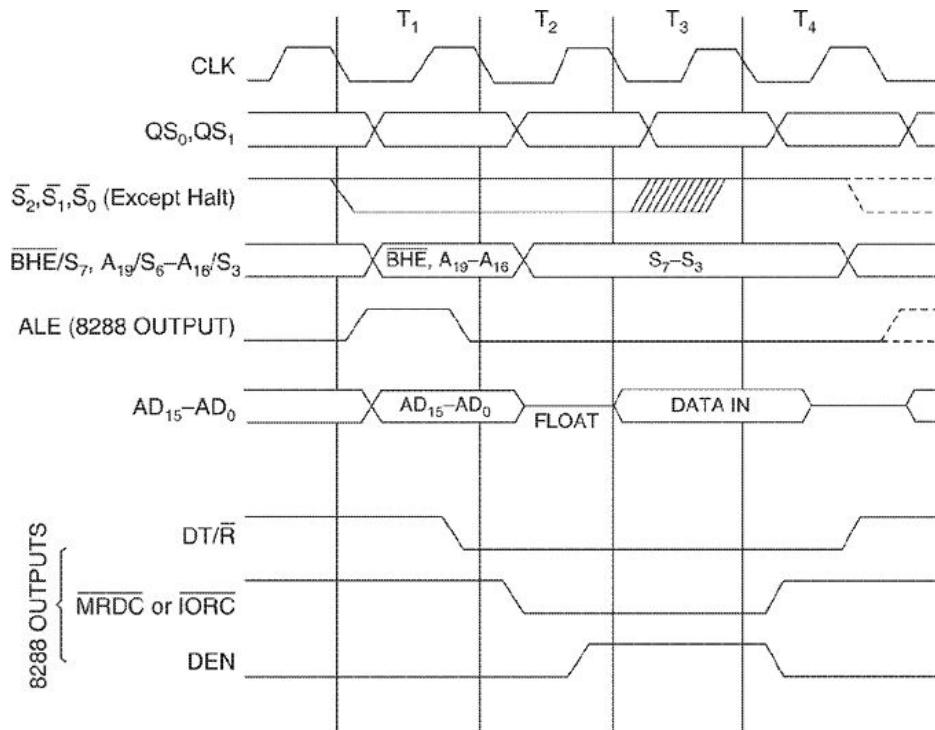


Figure 5.31 Memory read bus cycle in maximum mode.

T₁

- $\overline{S_0}, \overline{S_1}, \overline{S_2}$ are set by the 8086 in the beginning of clock cycle. It is decoded by the 8288 bus controller.
- ALE is pulsed high.
- \overline{BHE} is made high/low depending on 8/16-bit read at odd/even address boundary.
- $\overline{DT/R}$ goes low.
- Address is put in address/data bus.
- ALE is pulsed low. The trailing edge of ALE is used to latch the address information from address/data bus.

T₂

- \overline{BHE} is pulsed high, if made low in T₁.

- DEN goes high to enable transceiver.
- $\overline{\text{MRDC}}$ goes low as memory read control signal.

T₃

- Data is put by memory in address/data lines.
- Status lines $\overline{S_0}, \overline{S_1}, \overline{S_2}$ become inactive.

T₄

- $\overline{\text{MRDC}}$ goes high.
- DEN goes low.
- DT/R goes high.

The wait state operation is same as in the case of minimum mode. The READY signal is sampled at the end of T₂. If it is low, wait states are inserted between T₃ and T₄.

Note: For I/O read, all the signal activities are same as above except that $\overline{\text{MRDC}}$ is replaced by $\overline{\text{IORC}}$.

5.7.2 Memory Write Bus Cycle

The maximum mode memory write bus cycle is shown in Figure 5.32. The signal activities during different clock periods are as follows:

T₁

- $\overline{S_0}, \overline{S_1}, \overline{S_2}$ are set by the 8086 in the beginning of clock cycle and are decoded by 8288.
- ALE is pulsed high.
- $\overline{\text{BHE}}$ is made high/low depending on 8/16-bit write on odd/even address boundary.
- Address is put on address/data bus.
- ALE is pulsed low. The trailing edge of ALE is used to latch the address information from address/data lines.

T₂

- $\overline{\text{BHE}}$ is pulsed high, if made high in T₁.
- DEN goes high to enable transceiver.

T₃

- $\overline{\text{MWTC}}$ goes low as memory write control signal.
- Data is put on address/data lines by the 8086.
- Status lines $\overline{S_0}, \overline{S_1}, \overline{S_2}$ become inactive.

T₄

- $\overline{\text{MWTC}}$ goes high
- DEN goes low

Note: For I/O write operation the signal corresponding to $\overline{\text{MWTC}}$ is $\overline{\text{IOWC}}$. The rest of the signals are same.

The write operation can also be performed by write control signals $\overline{\text{AMWTC}}$ (for memory write) and $\overline{\text{AIOWC}}$ (for I/O write). Figure 5.32 also shows the signal activities during memory write bus cycle using $\overline{\text{AMWTC}}$ signal. You will notice that the signal $\overline{\text{AMWTC}}$ is activated from T₂ to T₄, i.e. one clock cycle earlier than $\overline{\text{MWTC}}$. In case of I/O write bus cycle using $\overline{\text{AIOWC}}$, all signals in Figure 5.32 remain same except $\overline{\text{AMWTC}}$ which is replaced by $\overline{\text{AIOWC}}$.

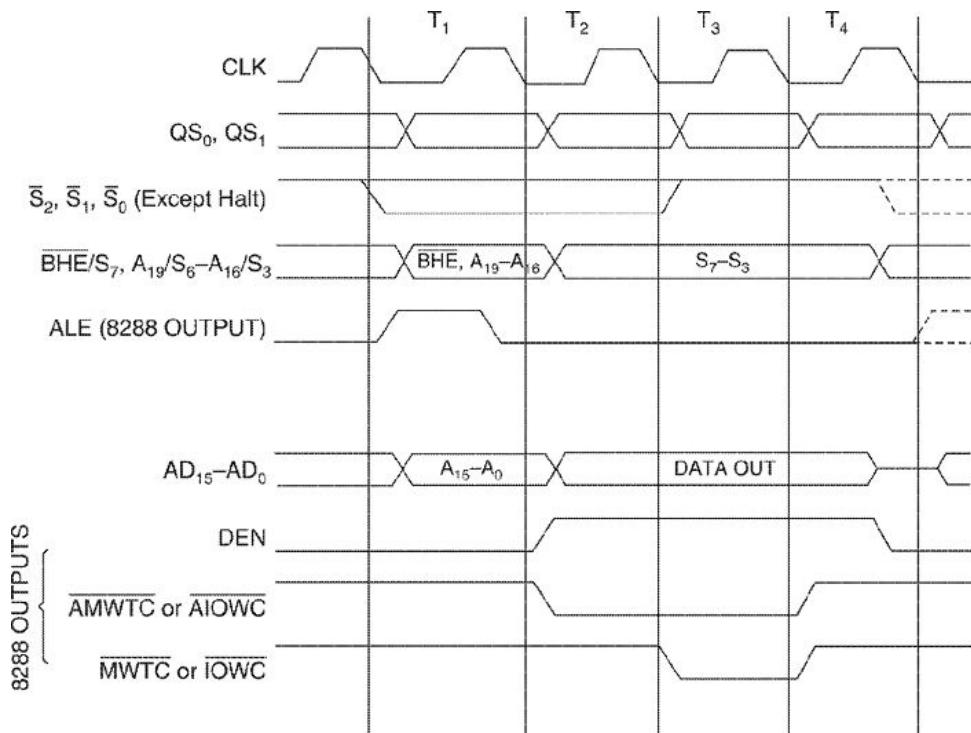


Figure 5.32 Memory write bus cycle in maximum mode.

In case of the 8088 memory read, I/O read, memory write and I/O write cycles in maximum mode, BHE is absent and the address/data bus AD₀-AD₁₅ is replaced by the address/data bus AD₀-AD₇ and the address bus A₈-A₁₅. The rest of the signal timings are almost same as those of the corresponding bus cycle in the 8086.

5.8 INTEL 8086 SYSTEM CONFIGURATIONS

Having dealt with Clock Generator Intel 8284, Bus Transceiver Intel 8286 and Bus Controller Intel 8288 chips, let us now see how they may be connected to the 8086 to evolve some simple system configurations.

Figure 5.33 shows the interconnection between the 8286 bus transceiver, the 8284 clock generators, the 8282 latch and the 8086 microprocessor. The 8282 latch is used to demultiplex the address/data bus lines to two separate address and data buses. The ALE (Address Latch Enable) signal acts as strobe for latching the address from address/data lines. As described earlier, it is the trailing edge of ALE which triggers the latching operation.

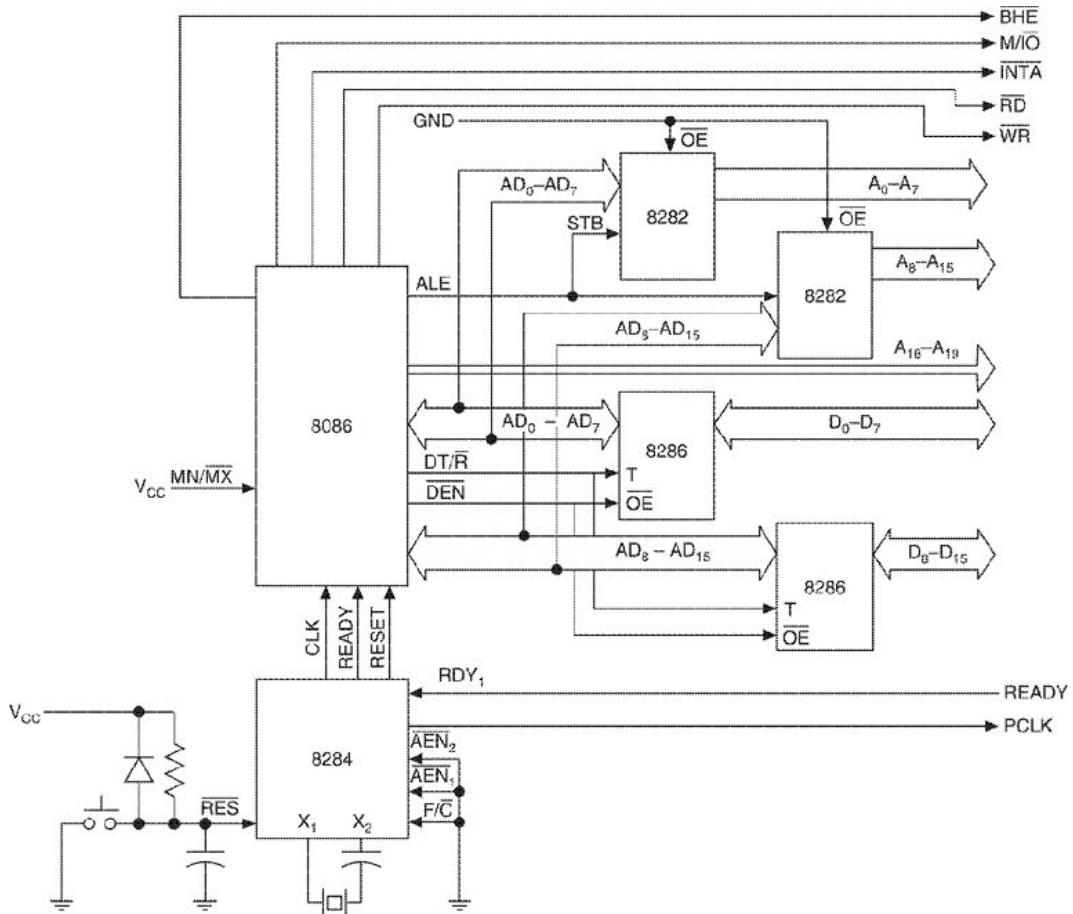


Figure 5.33 The 8086 system configuration in minimum mode.

The 8284 clock generator provides CLK, READY and RESET signals to the 8086. CLK is generated based on the frequency of crystal connected between X₁ and X₂.

The READY signal, in synchronization with CLK, is generated by the 8284 based on input at RDY₁ pin. The RDY₂ pin has not been used in the configuration.

For resetting, the 8086 requires that the input at RESET pin remains high for at least 50 ms. The RC network circuit at RES pin of the 8284 ensures that when the manual reset switch is closed, the low level remains at RES for at least 50 ms, due to large time constant of the RC network. Since this is a single microprocessor-based configuration, AEN₁, AEN₂ and F/C are not used and therefore grounded.

The data bus D₀ to D₁₅ is bidirectional. In order to create a separate data bus from address/data bus, two Intel 8286 bidirectional bus transceivers are used. DT/R and DEN output of the 8086 are connected to the 8286 as T and OE respectively. The system configuration in Figure 5.33 corresponds to minimum mode operation of the 8086; MN/MX pin is at high level to support this.

The system configuration for maximum mode operation can be evolved from minimum mode operation by using the bus controller 8286 and grounding the $\overline{\text{MN/MX}}$ pin, as shown in Figure 5.34.

The ALE, $\overline{\text{DT/R}}$ and DEN are now generated by the 8288 bus controller.

The IOB pin is grounded indicating that the 8288 is operating in system bus mode in which all signals are active.

$\overline{S_0}, \overline{S_1}, \overline{S_2}$, status signals of the 8086, are input to the 8288, which decodes them and generates $\overline{\text{INTA}}$, $\overline{\text{MRDC}}$, $\overline{\text{MWTC}}$, $\overline{\text{IORC}}$, $\overline{\text{IOWC}}$, $\overline{\text{AMWC}}$ and $\overline{\text{AIOWC}}$. These signals are used for interfacing the 8086 to memory and I/O devices as discussed earlier. We shall use some of these again, while dealing with memory interfacing with the 8086.

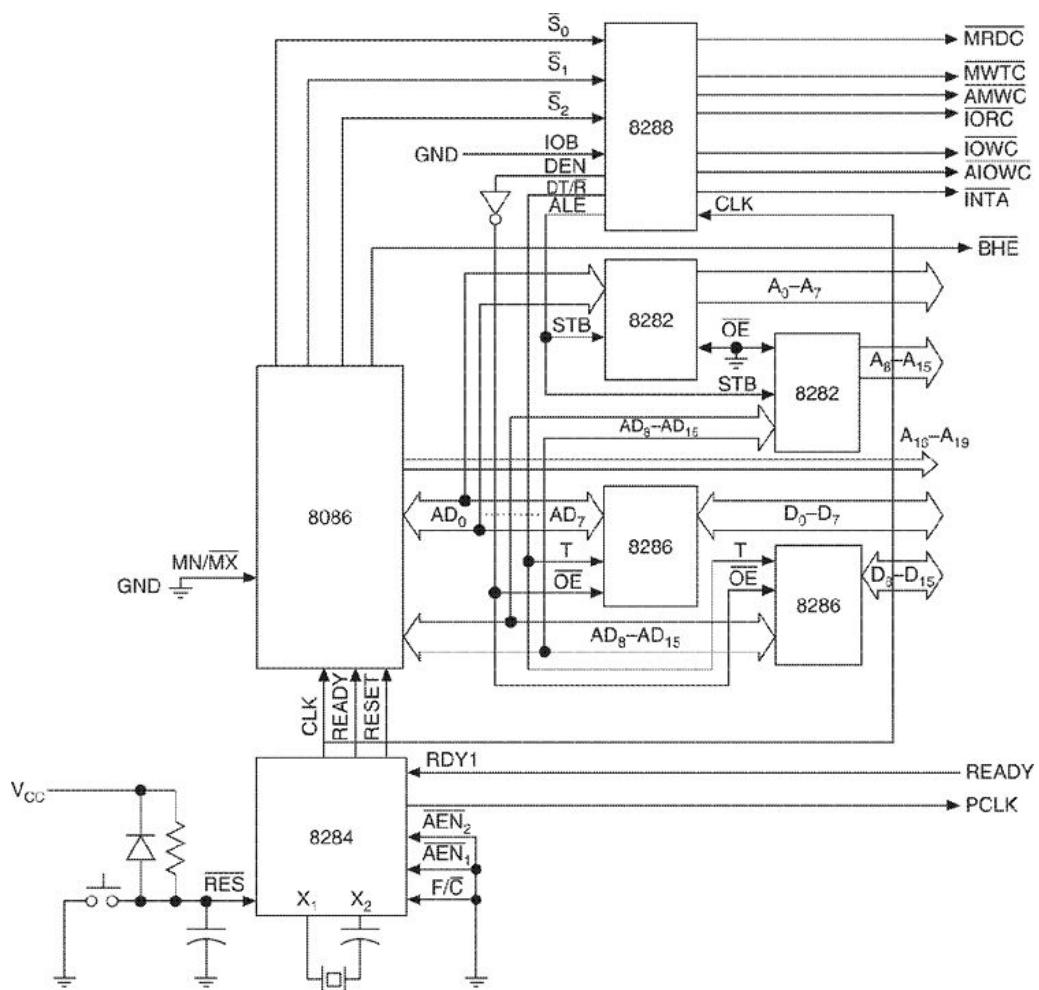


Figure 5.34 The 8086 system configuration in maximum mode.

5.9 MEMORY INTERFACING

We shall now discuss the interfacing of the 8086 microprocessor to memory as it is the main step in the design of a microprocessor-based

system. Since the 8086 has 20 address lines, a total of 1MB memory space can be interfaced. Various types of memory chips such as RAM, ROM, EPROM, etc. may be connected within this space.

The number of memory chips and the address space used will depend on the application needs. It is not necessary that the address space of any microprocessor system should start only from 00000. It may start at any boundary of 1 KB, i.e. the address bits A_0 to A_9 must be zero at the starting address. Even this restriction is for convenience sake only and can be overcome by deep decoding.

The first task is to allocate the address range to different memory chips and devise the address decoding circuit so that based on address information on address lines, the chip select signal to select the particular memory chip may be generated. After the selection of the memory chip, the read/write operation may be performed.

The block diagram of address decoder 74LS138 which is widely used is shown in Figure 5.35. The decoder is selected when $G_1 \cdot (\overline{G_2B} \cdot \overline{G_2A}) = 1$. On selection, the decoder generates the signals Y_0 to Y_7 based on the information available at A, B and C pins, and based on the truth table shown in Figure 5.36.

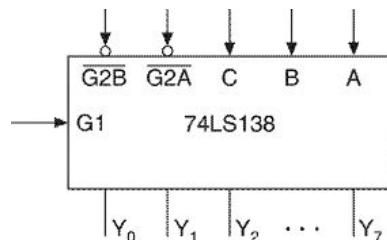


Figure 5.35 Block diagram of 74LS138 decoder.

Inputs			Outputs							
C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

Figure 5.36 Truth table of 74LS138 decoder.

Suppose a 2 KB chip 2716/6116 is to be interfaced to the 8086 starting from 00000 to 007FFH. If you examine the address information, you will find that bits A₁₉ to A₁₁ in address information will always be zero. Thus, we may connect address lines to 74LS138 pins in the following manner:

$$A = A_{11}, B = A_{12}, C = A_{13},$$

$$\begin{aligned}\overline{G2A} &= A_{14}, \quad \overline{G2B} = A_{15} \\ G1 &= (\overline{A_{16}} \cdot \overline{A_{17}} \cdot \overline{A_{18}} \cdot \overline{A_{19}}) \cdot (M/\overline{IO})\end{aligned}$$

Since A, B and C are zero, Y₀ will be low which can be directly used as active low chip select (\overline{CS}) signal for memory chip. Figure 5.37 explains the decoding strategy outlined above. For the next address range which starts from 00800H, A₁₁ bit will become 1. Thus, the pin A of 74LS138 decoder will become 1 for the address range starting from 00800H and this will make Y₁ low which can be used to select the next memory chip as shown in Figure 5.37.

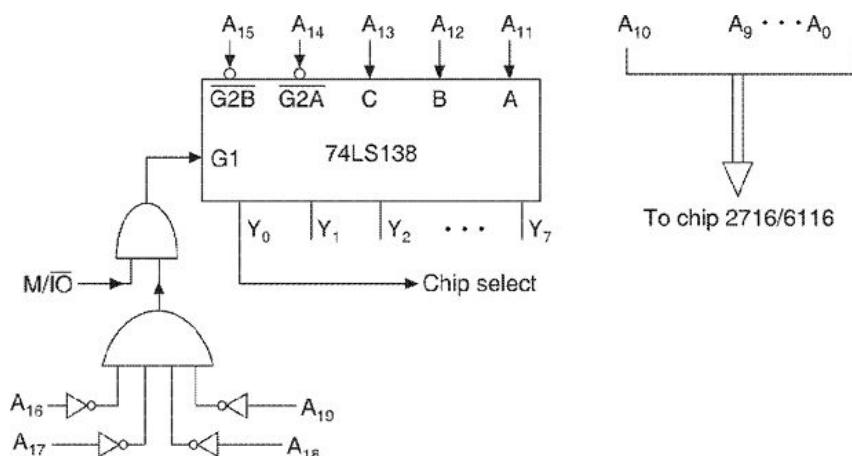


Figure 5.37 Generation of chip select signal for memory.

Let us now work out the above decoding strategy again in the following manner:

$$\begin{aligned}G1 &= M/\overline{IO} \\ \overline{G2A} &= (A_{14} \text{ OR } A_{15} \text{ OR } A_{16}) \\ \overline{G2B} &= (A_{17} \text{ OR } A_{18} \text{ OR } A_{19}) \\ A &= A_{11}, \quad B = A_{12}, \quad C = A_{13}\end{aligned}$$

You will find that not only the above but many combinations of the above may work well as effective decoding strategies.

Let us now take another example in which the 8 KB memory chip 2764/6164 is to be interfaced starting from address FA000H. Bit patterns of starting address FA000H and end address FBFFFH are

	A ₁₉	—	A ₁₅	—	A ₁₁	—	A ₇	—	A ₃	—	A ₀
FA000H =	1	1	1	1	1	0	1	0	0	0	0
FBFFFH =	1	1	1	1	1	0	1	1	1	1	1

Bits which do not undergo change are A₁₄ to A₁₉. Thus these bits may be utilized for G₁, $\overline{G2A}$ and $\overline{G2B}$.

Bits A₁₂, A₁₃, A₁₄ may be connected to A, B, C input pins respectively. Both Y₂ and Y₃ outputs of 74LS138 will get activated in the above address range. Thus Y₂ AND Y₃ = \overline{CS} of memory chip 2764/6164. There may be different combinations for G₁, $\overline{G2A}$ and $\overline{G2B}$ as mentioned before. One possible combination is

$$\begin{aligned} G1 &= M/\overline{IO} \\ \overline{G2A} &= \overline{A_{19}} \text{ OR } \overline{A_{18}} \\ \overline{G2B} &= \overline{A_{16}} \text{ OR } \overline{A_{17}} \end{aligned}$$

However, as discussed earlier, the 8086 memory is organized into two memory banks. Memory bank-1, containing all even addresses, is called the low-order memory bank, whereas memory bank-2, containing all odd addresses, is known as the high-order memory bank. The low-order memory bank is selected by $\overline{A0}=1$ whereas for high-order memory bank addresses (i.e. odd addresses), the 8086 outputs an active low signal \overline{BHE} . Thus, the address decoding scheme mentioned earlier needs to be modified in the following manner:

- (a) A₀ is not used for address decoding but for memory bank selection along with \overline{BHE} .
- (b) Memory is now organized into two distinct parts—chip for high-order address bank and chip for low-order address bank.

This may be done in two ways:

- Separate decoder for high- and low-order banks.
- Single decoder for both the banks.

Figure 5.38 shows memory address decoding using two separate decoders. Address lines A₁ to A₁₁ are connected to memory chips. The

address range is 00000 to 00FFFF, i.e. total 4 KB. The decoding is performed as

$$A = A_{12}, \quad B = A_{13}, \quad C = A_{14}$$

$$\overline{G2A} = A_{15} \text{ OR } A_{16} \text{ OR } (M/\overline{IO})' = (\overline{A}_{15} \cdot \overline{A}_{16} \cdot (M/\overline{IO}))'$$

$$\overline{G2B} = A_{17} \text{ OR } A_{18} \text{ OR } A_{19} = (\overline{A}_{17} \cdot \overline{A}_{18} \cdot \overline{A}_{19})'$$

For lower decoder, $G1 = \overline{A_0}$

For upper decoder, $G1 = BHE$

Thus the upper memory chip 2716/6116 (2 KB) will be part of high-order memory bank whereas the lower memory chip 2716/6116 (2 KB) will be part of low-order memory bank. Other output lines of decoders, i.e. Y_1 to Y_7 can be used to connect similar chips. Thus the total memory capacity of 32 KB may be interfaced in this way.

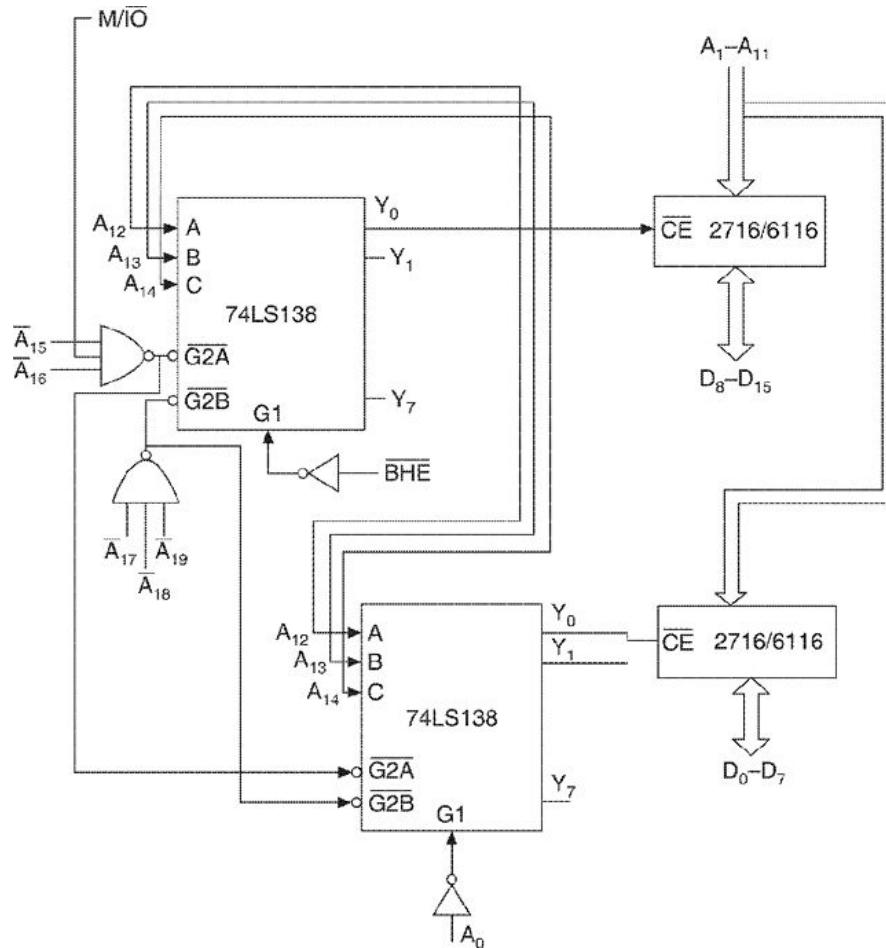


Figure 5.38 Memory address decoding (separate decoder system).

Since apart from BHE and A_0 , both the decoders have the same input signals, we may, therefore, explore whether it is possible to use only one address decoder.

Figure 5.39 describes the memory address decoding using one 74LS138 decoder. In this scheme

$$\begin{aligned} G1 &= M/\overline{IO} \\ \overline{G2B} &= [\bar{A}_{17} \cdot \bar{A}_{18} \cdot \bar{A}_{19}]' \\ \overline{G2A} &= [\bar{A}_{15} \cdot \bar{A}_{16}]' \end{aligned}$$

Y_0 is combined with \overline{BHE} to select the high-memory bank chip, and Y_0 is combined with A_0 to select the low-memory bank chip.

In addition, address decoding for eight 16-bit I/O ports is shown using the 74LS138 decoder.

$$\begin{aligned} A &= A_0, B = A_1, C = A_2 \\ \overline{G2A} &= \overline{G2B} = GND \\ G1 &= (M/\overline{IO})' \end{aligned}$$

It is evident that the decoder will be selected when M/\overline{IO} is low, i.e. only for I/O operations, and depending on the information at A_0, A_1, A_2 bits, one of the eight I/O ports will be selected.

We have purposely not shown the connections of read/write control signals to memory chips and I/O ports, as these are different in minimum and maximum modes. In minimum mode, \overline{RD} and \overline{WR} control the read/write operations whereas in maximum mode, control signals \overline{MRDC} , \overline{MWTC} , \overline{IORTC} , \overline{IOWC} , \overline{AMWC} and \overline{AIOWC} are used for read/write operations. \overline{INTA} is used for memory write operation in interrupt cycle. We shall now discuss minimum and maximum mode system configurations with memory and I/O interface.

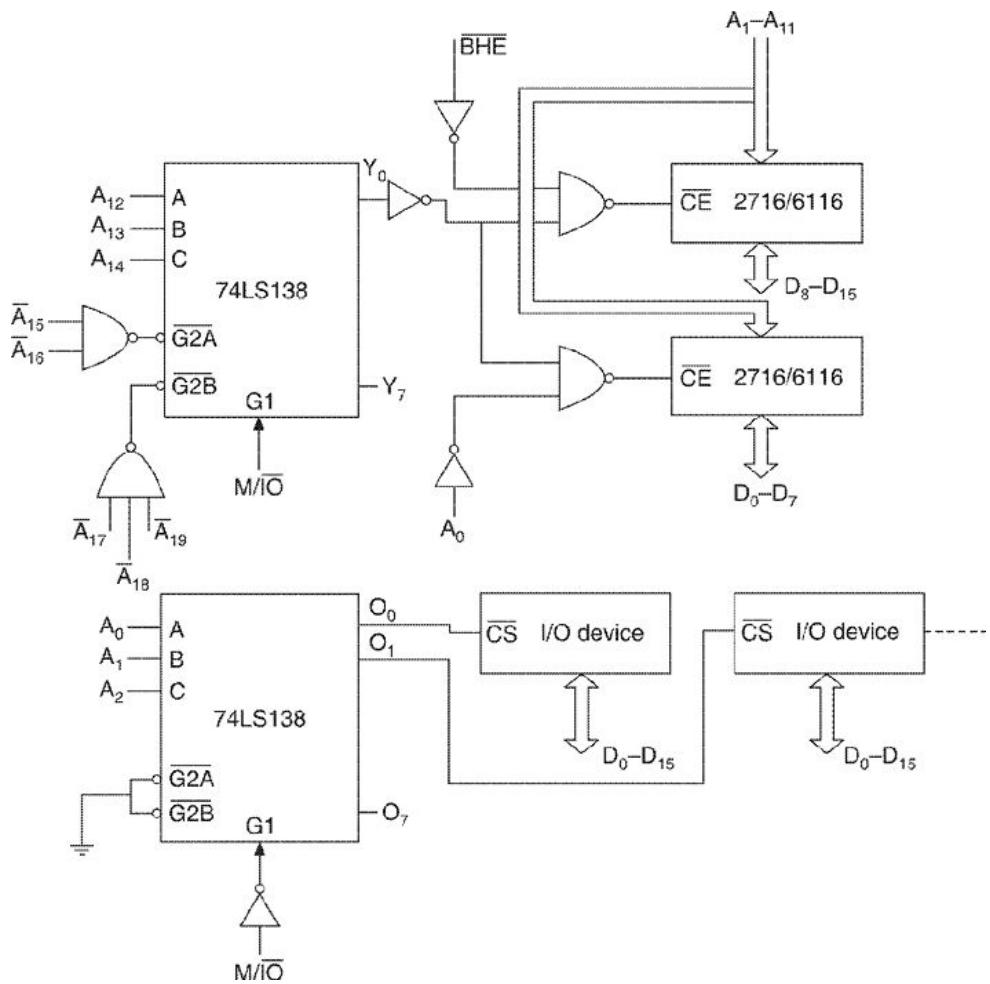


Figure 5.39 Memory and I/O address decoding (single decoder scheme).

5.10 MINIMUM MODE SYSTEM CONFIGURATION

The 8086 based system configuration in minimum mode having both memory and I/O interface is shown in Figure 5.40. It is basically a combination of minimum mode configuration (as shown in Figure 5.33) and memory and I/O interface (as shown in Figure 5.39). For read/operation on memory, \overline{RD} from the 8086 is connected to \overline{OE} pin of memory chip whereas \overline{WR} from the 8086 is connected to \overline{WR} pin of memory chip. For I/O devices, pins for \overline{RD} and \overline{WR} signals will be available and should be connected. The system has clock generator Intel 8284, latch Intel 8282 and Intel 8286 transceiver chips whose function have already been explained.

5.11 MAXIMUM MODE SYSTEM CONFIGURATION

The 8086 system configuration in maximum mode shown in Figure 5.41 is the combination of maximum mode configuration having basic chips as in Figure 5.34 and memory and I/O interface as in Figure 5.39. The M/IO is not generated in maximum mode. Instead, separate read and write control signals for both memory and I/O devices are generated. Therefore the G₁ pins in both the 74LS138 decoders are connected to V_{CC}.

For memory read/write operation, \overline{MRDC} and \overline{MWTC} from the 8288 bus controller are connected to \overline{OE} and \overline{WR} pins, respectively, of memory chips. For I/O read/write, \overline{IORC} and \overline{IOWC} from the 8288 bus controller may be connected to similar pins (\overline{RD} and \overline{WR} in diagram) of the I/O device. We may, however, use advanced memory (\overline{AMWC}) and I/O (\overline{AIOWC}) write control signals instead of normal write control signals. The functions of other chips, i.e. the 8284 clock generator, the 8282 latch, the 8286 transceiver and the 8288 bus controller have been explained previously.

5.12 INTERRUPT PROCESSING

The utility and processing of interrupts by microprocessor have been dealt with in detail in Chapter 2. The 8086 has four sources of interrupts:

- Software or within-program logic
- Single step condition
- External logic as a non-maskable interrupt
- External logic as a maskable interrupt

Central to interrupt processing is the logic using which the location of Interrupt Servicing Routine (ISR) for a particular interrupt is determined. In the 8085 interrupt system, the locations of Interrupt Servicing Routines for RST 5.5, RST 6.5, RST 7.5 and TRAP interrupts are already fixed by hardware, whereas for INTR the external logic inputs the location of ISR through ‘Call addr’ or ‘RST n’ instructions.

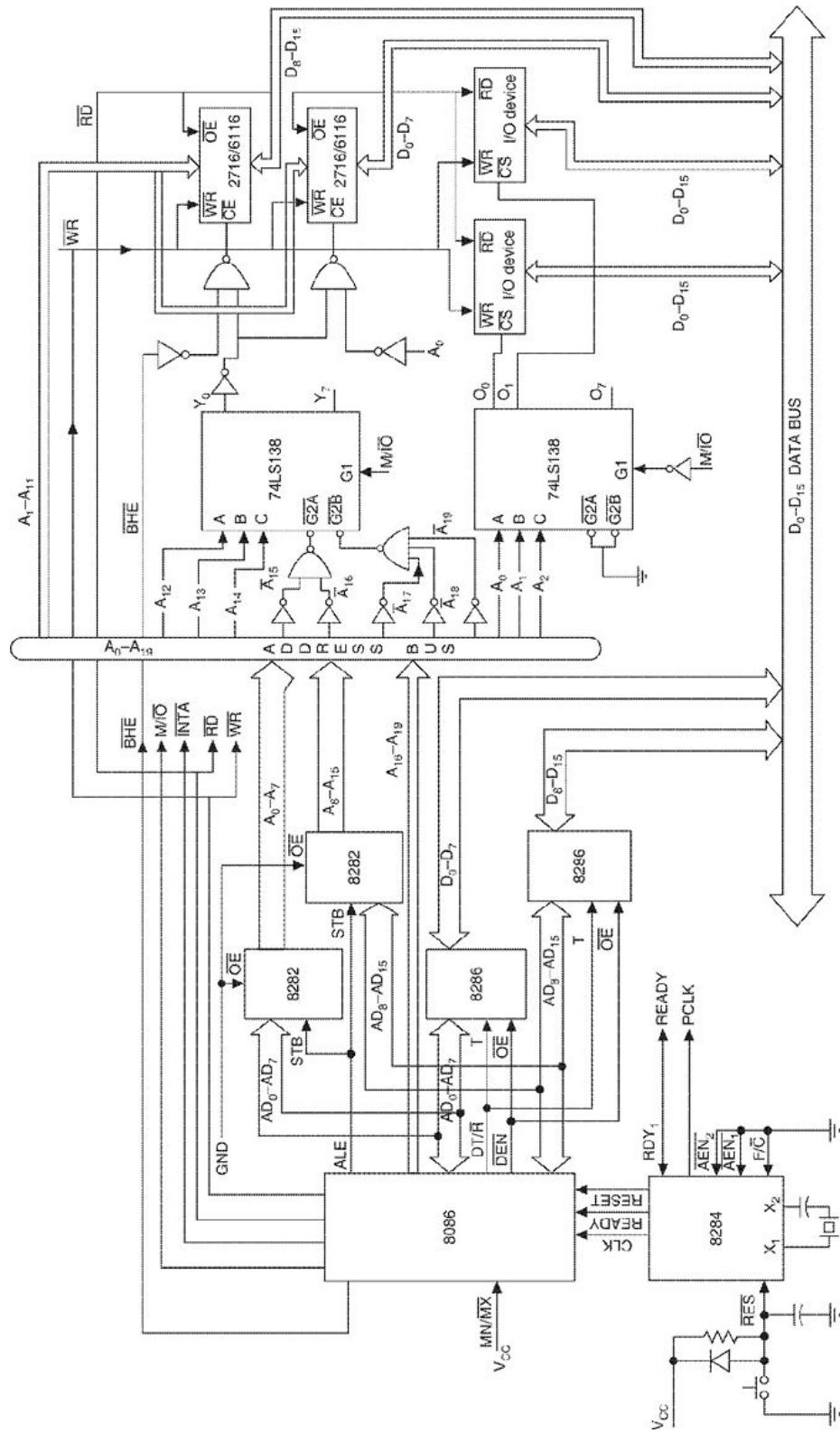


Figure 5.40 The 8086 minimum mode configuration with memory and I/O interface.

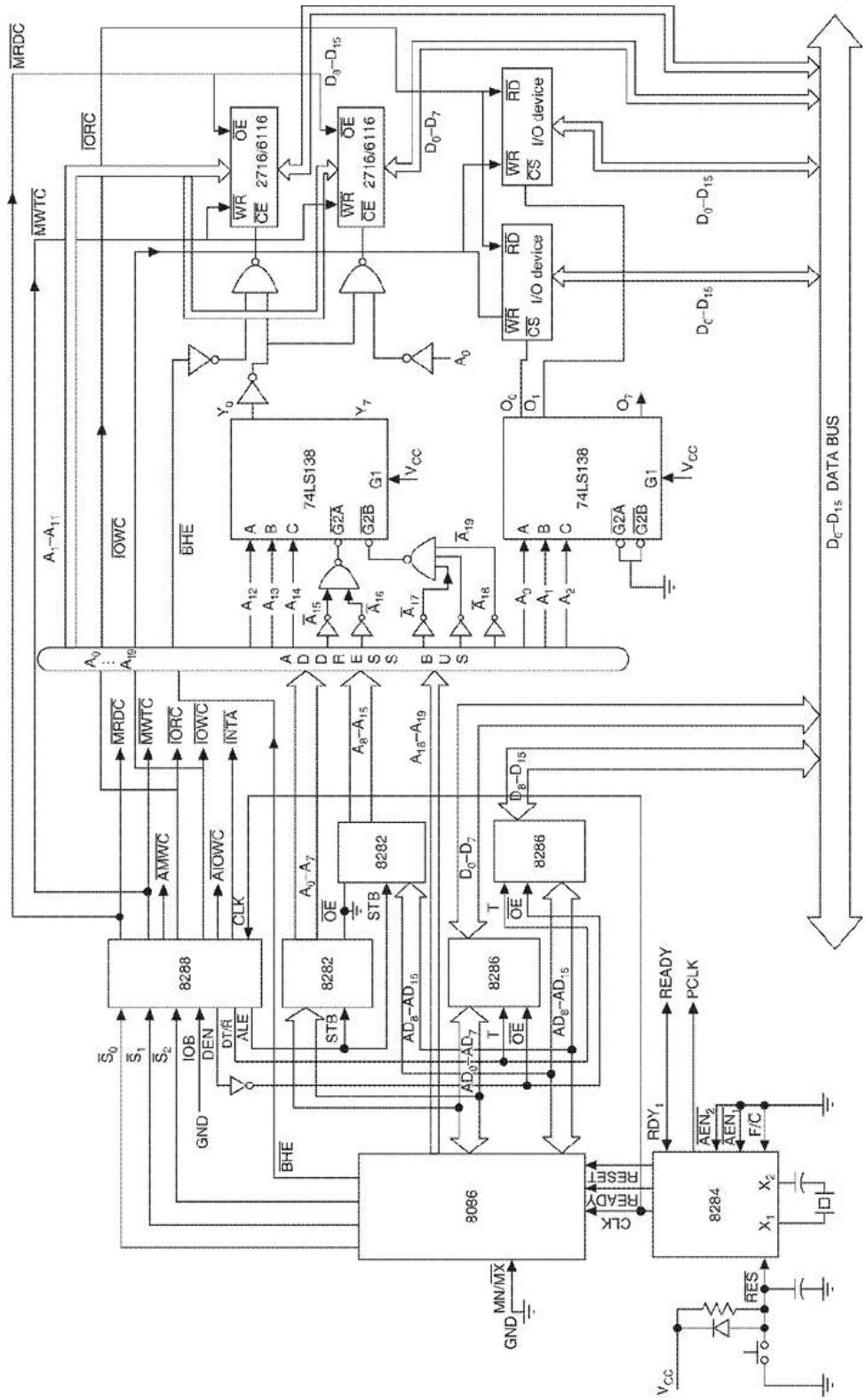


Figure 5.41 The 8086 maximum mode configuration with memory and I/O interface.

The 8086 microprocessor maintains an Interrupt Vector Table which stores the information regarding the location of interrupt service routines of various interrupts. In a very simple way, whenever an interrupt occurs, the memory location of ISR is determined using the vector table, and the

program control branches to ISR after saving the flags and the program location (Instruction Pointer and Code Segment Register) in stack. The IRET (Return from Interrupt Service Routine) is the last instruction of ISR. When it occurs the 8086 pops and restores the contents of flags, Instruction Pointer and Code Segment Register and the execution of the main program restarts from the place where it was interrupted. The interrupt vector table can be up to 1 KB starting from 00000H to 003FFH of main memory. For each interrupt, the following is stored:

- Code Segment Register (16 bits)
- Instruction Pointer (16 bits)

These two determine the memory location, i.e. where the ISR for the interrupt is located. The total 256 numbers of interrupts may be represented (each in 4 bytes) in the vector table. These are identified as Type 0 to 255 or Vector 0 to 255 interrupts. The 8086 vector table is shown in Figure 5.42.

Memory location	Vector/type
3FFH	(Available) — 255
3FCH	⋮
084H	(Available) — 33
080H	(Available) — 32
07FH	(Reserved) — 31
⋮	⋮
014H	(Reserved) — 05
010H	Overflow — 04
00CH	1-byte int instruction — 03
008H	Non-maskable — 02
004H	Single-step — 01
	Divide error — 00
000H	16 bits
CS base address IP offset	

Figure 5.42 The 8086 interrupt vector table.

A number of the vector table entries are reserved to serve specific interrupts as follows:

Type/Vector	Memory address	Specific interrupt

	(in Hex)	
0	00000	Divide by Zero
1	00004	Single-Step Mode
2	00008	Non-Maskable Interrupt
3	0000C	INT Software Interrupt (default option)
4	00010	INTO Software Interrupt

A number of entries (Total 27—Vector/Type 5 to 31) are reserved by Intel. The remaining 224 (32 to 255) vectors are available for user-defined interrupts. The external logic passes the vector/type number of the interrupt to the 8086 during interrupt acknowledge bus cycle. The type/vector number is multiplied by 4 to get the memory location. As an example, if the type/vector number of the interrupt is 32, its memory address in vector table will be $32 \times 4 = 128 = 00080H$.

5.12.1 Software Interrupts

A software interrupt request may occur when the following operations happen.

- (a) When divide-by-zero occurs. During the execution of a program, if the quotient from either DIV or IDIV instruction is too large to fit in the quotient register, the 8086 will generate the Divide-by-Zero interrupt request. Since the 8086 response is automatic, this interrupt cannot be disabled.
- (b) Execution of the INT instruction ‘INT n’ ($n = 0$ to 255) may be used to cause the 8086 to branch to any of the 256 interrupt types mentioned in the interrupt vector table. For example, INT 40 will branch to ISR specified by Vector/Type 40 (memory location $160 = A0H$) in the interrupt vector table. One of the possible uses of this generalized software interrupt caused by ‘INT n’ instruction is to test the different Interrupt Servicing Routines. For example, ‘INT 0’ causes the execution of ISR of the Type/Vector 0, i.e. Divide-by-Zero interrupt. Similarly, NMI ISR may be tested by the INT 2 instruction.

INT 3 is a special case of this software interrupt and is used for inserting breakpoints for debugging the program. When we insert a breakpoint at any place in our program, the 8086 executes the program up to the breakpoint and then branches to breakpoint ISR.

We may write the breakpoint ISR to display or store the contents of the various registers/memory locations for debugging the program.

(c) Execution of an Interrupt on Overflow (INTO), if the Overflow flag is set. Overflow is set when the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location. As an example, let us perform addition on two 8-bit numbers 120 and 70. In binary, the operation will look like as follows:

	7	6	5	4	3	2	1	0
120 =	0	1	1	1	1	0	0	0
70 =	0	1	0	0	0	1	1	0
	1	0	1	1	1	1	1	0

The result in signed arithmetic is a negative number, that is, -68. Such conditions need to be guarded while developing the program. There are two ways in which such conditions may be detected and then corrected. One way is to use Jump on Overflow (JO) instructions and the other is to use Interrupt on Overflow (INTO) instructions immediately after signed addition or subtraction. If no overflow condition occurs, these instructions work as No Operation (NOP) instruction. When the overflow condition occurs, Jump on Overflow (JO) instructions will branch the program control to the address mentioned in the instructions.

On the other hand, if overflow occurs and the INTO instruction is used, the 8086 will execute Interrupt on Overflow software interrupt at Type/Vector 4 of the interrupt vector table.

When any of the software interrupts occurs and is acknowledged, the 8086 takes the following steps:

1. The flag register contents are pushed onto the stack. The stack pointer is decremented by 2.
2. The flag register contents are cleared. This disables the single step logic and maskable interrupt INTR.
3. The CS (Code Segment) register contents are pushed onto the stack. The stack pointer is decremented by 2.
4. The IP (Instruction Pointer) register contents are pushed onto the stack. The stack pointer is decremented by 2.
5. The new CS register contents are taken from the appropriate interrupt vector location. As already discussed, with the exception of INT instructions, software interrupts have dedicated vector locations.
6. The new IP contents are taken from the interrupt vector location.

5.12.2 Single Step Interrupt

When TF (Trap Flag) is set, the 8086 executes the Single Step Mode Interrupt after the execution of each instruction. The Single Step Mode Interrupt occurs at Type/Vector 1 in the interrupt vector table. The 8086 takes the following steps when a Single Step Interrupt occurs.

1. The flag register contents are pushed onto the stack. The stack pointer is decremented by 2.
2. The flag register contents are cleared. This disables the maskable interrupt INTR and single step logic during the execution of ISR.
3. The CS (Code Segment) register contents are pushed onto the stack. The stack pointer is decremented by 2.
4. The IP (Instruction Pointer) register contents are pushed onto the stack. The stack pointer is decremented by 2.
5. The new CS register and IP register contents are taken from Type/Vector 1 of the interrupt vector table.

The ISR is written normally to debug the program. Usually the register contents and flags etc. are stored for later examination. As mentioned earlier, the last instruction of ISR is IRET, whose execution restores the flags, CS and IP register contents and thus the next instruction of the main program gets executed, and immediately the program branches to single step ISR. Thus, the Single Step ISR is executed after the execution of each instruction. Remember that TF flag is restored with the execution of the IRET instruction. We need to know another important aspect with respect to single stepping mode. This aspect relates to setting of Trap Flag. Note that the 8086 does not have an instruction to set or reset the Trap Flag. Thus this flag can be set by moving the flag register to stack or memory, changing it there and then restoring it in the flag register. A sample program is given below.

```
PUSHF           ; Push flags on stack
MOV BP, SP      ; Copy SP to BP to use as index
OR [BP+0], 0100H ; Set the bit 8 in the memory pointed by
                  ; BP,
                  ; i.e. set the Trap Flag bit
POPF            ; Restore the flag register with TF = 1
```

To reset the flag, the OR instruction will be replaced by AND [BP+0], FEFFH. Note that BP cannot be used as pointer without displacement,

that is why, [BP+0] has been used in OR instruction.

Also note that ISR for Single Step Mode Interrupt is not executed in a single step since single step logic is disabled by making TF = 0 during the acknowledgement.

5.12.3 Non-Maskable Interrupt

It is a rising-edge triggered interrupt, i.e. a low-to-high signal at the NMI pin will cause the interrupt to be recognized. The NMI interrupt has a lower priority than the software interrupts but has a higher priority over the maskable interrupt at the INTR pin. Thus, the NMI interrupt is accepted only when no software interrupt is being processed or is pending. Very much like the Divide-by-Zero interrupt, the location in vector table for the NMI interrupt is fixed, i.e. Type/Vector 2. When the NMI interrupt is recognized, the 8086 takes the following steps:

1. The flag register contents are pushed onto the stack. The stack pointer contents are decremented by 2.
2. The flag register contents are cleared. This disables maskable interrupt INTR and single step logic.
3. The CS register contents are pushed onto the stack. The stack pointer is decremented by 2.
4. The IP register contents are pushed onto the stack. The stack pointer is decremented by 2.
5. The new CS and IP register contents are taken from Type/Vector 2 of the interrupt vector table.

5.12.4 Maskable Interrupt

The maskable interrupt at INTR pin of the 8086 is high-level triggered. Thus a high-level at INTR pin will cause the acceptance of interrupt request by the 8086. The interrupt at INTR pin can be enabled (IF = 1) by the instruction STI or can be disabled (IF = 0) by the instruction CLI. When the 8086 is reset, the INTR is disabled since the interrupt flag IF is cleared. When the interrupt request at INTR pin is accepted, the 8086 performs the following functions.

1. Two interrupt acknowledgement bus cycles are executed as shown in Figure 5.43. The purpose of these bus cycles is to get the interrupt type/vector from the external device. In the first

interrupt acknowledgement bus cycle, the 8086 floats the data bus lines, AD₀–AD₁₅, and sends out an INTA pulse. Note that in maximum mode it is the bus controller Intel 8288 which issues the INTA signal. This indicates that an interrupt acknowledgement cycle is in progress and the system is ready to accept the type/vector information from the external device.

2. The external device must send back a byte of data on lines AD₀–AD₇ in response to the INTA pulse during the second interrupt acknowledgement cycle. This data is multiplied by 4 to get the vector information on the interrupt.
3. The flag register contents are pushed onto the stack. The stack pointer contents are decremented by 2.
4. The flag register contents are cleared. This disables further maskable interrupts and the single step logic.
5. The CS register contents are pushed onto the stack. The stack pointer contents are decremented by 2.
6. The IP register contents are pushed onto the stack. The stack pointer contents are decremented by 2.
7. The new CS and IP register contents are taken from the vector location calculated earlier in Step 2.

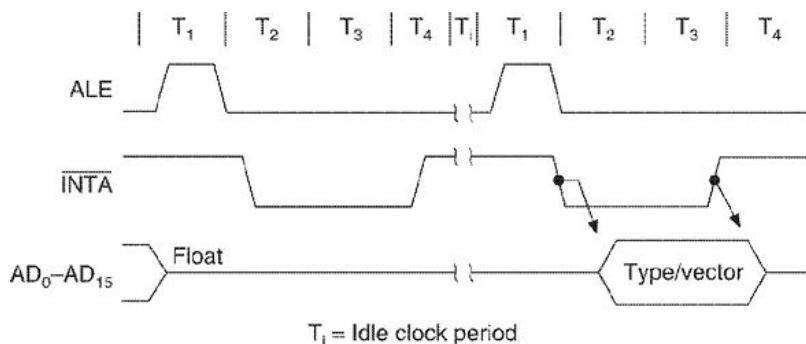


Figure 5.43 Interrupt acknowledgement for maskable interrupt.

5.12.5 Interrupt Priorities

The priority structure for various interrupts in the 8086 is as follows:

<i>Interrupt</i>	<i>Priority</i>
Divide-by-Zero, INT n and INTO	Highest
NMI	
INTR	
Single Step	Lowest

When two interrupt requests are encountered at the same time, the interrupt with the higher priority is serviced first. Thus if INTO (Interrupt on Overflow) and INTR (Maskable Interrupt) occur at the same time, INTO will be serviced first since it has a higher priority. During the servicing of INTO, the 8086 clears the IF flag thus disabling the INTR interrupt. On completion of interrupt servicing, the IRET instruction restores the flags. This enables the INTR interrupt which is taken up for servicing.

Another interesting case is when Divide-by-Zero interrupt and NMI (Non-Maskable Interrupt) occur at the same time.

Since the 8086 checks for internal interrupts before it checks for NMI, the Divide-by-Zero interrupt is taken up first. The 8086 pushes the flags, IP and CS contents on to the stack and executes the ISR to service the Divide-by-Zero interrupt. Note that NMI cannot be disabled and thus the 8086 recognizes NMI. It pushes the contents of flags, CS and IP on to the stack and executes ISR for NMI. On completion, the IRET instruction restores flags, IP and CS, and Divide-by-Zero will be serviced. On completion of Divide-by-Zero ISR, the IRET instruction restores the flags, and IP and CS, and the control returns to the main program. Thus NMI gets serviced in between the ISR for Divide-by-Zero.

5.13 DIRECT MEMORY ACCESS

In direct memory access, the external logic is allowed to access the memory directly and to perform the read/write operation. The 8086 floats its address/data lines during this period. The direct memory access operation has been described in Chapter 2. We shall discuss as to how it is implemented in the 8086 in minimum and maximum modes.

DMA in minimum mode ($MN/\overline{MX} = 5V$)

Very much like the 8085, the direct memory access in minimum mode of the 8086 is facilitated through HOLD and HLDA signals.

- When the external logic wishes to access the memory directly, it makes a request through high signal at HOLD pin.
- The 8086 samples the HOLD input at low-to-high transition of CLK.
- The 8086 acknowledges the HOLD request through high-level at HLDA (HOLD Acknowledge) pin at the end of the current bus

cycle, or during an idle clock period (if no bus cycle is being executed).

- The 8086 floats the address, data and control lines at the same time.
- The external logic accesses the memory and on completion, it withdraws the HOLD request by making the signal at HOLD pin low.
- The 8086 is sampling HOLD at every low-to-high transition of CLK. On detecting low signal at HOLD, it will make the signal at HLDA low.

Figure 5.44 illustrates the operation.

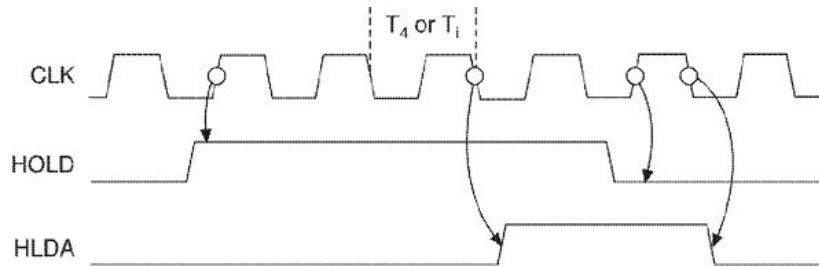


Figure 5.44 Timing diagram for direct memory access (minimum mode).

DMA in maximum mode ($MN/MX = GND$)

In maximum mode, two signals $\overline{RQ/GT_0}$ and $\overline{RQ/GT_1}$ are dedicated for the direct memory access operation. In this case, the DMA request and the DMA acknowledgement are received on the same line. Thus, $\overline{RQ/GT_0}$ and $\overline{RQ/GT_1}$ are two channels for DMA, using which, two separate external logics may access the memory directly. The DMA operation using $\overline{RQ/GT}$ is illustrated in Figure 5.45.

- The external logic makes a request for DMA by sending a low pulse at $\overline{RQ/GT}$. The 8086 samples $\overline{RQ/GT}$ at low-to-high transition of CLK. It is represented as ① in Figure 5.45.
- The 8086 acknowledges the DMA request at the end of the current bus cycle or an idle clock-period (if no bus cycle is in progress) by sending a low pulse on the same $\overline{RQ/GT}$ pin. The 8086 floats the address/data and control lines at the same time. It is shown as ② in Figure 5.45.
- On completion of memory operation, the external logic informs the 8086 by sending a low pulse on the same $\overline{RQ/GT}$ pin. The 8086 samples the pin on low-to-high transition of CLK and on finding

low signal, it regains the control of lines. It is shown as ③ in Figure 5.45.

An important question arises here! Does the 8086 stop the operation when the external logic is doing direct memory access? Obviously, the 8086 may execute the next instruction from the instruction queue, if it does not involve bus operation. It is true for both minimum as well as maximum modes.

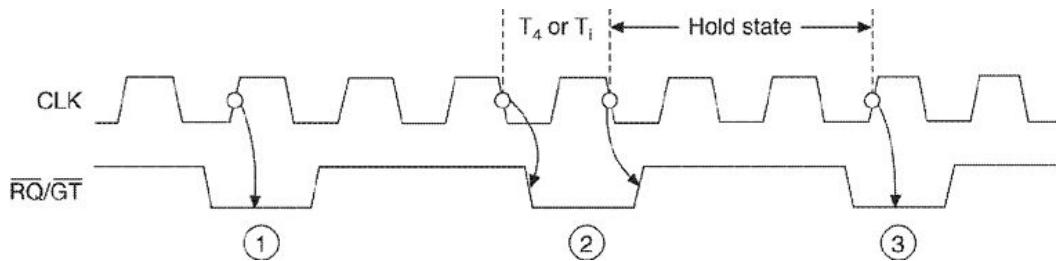


Figure 5.45 Timing diagram for direct memory access (maximum mode).

If the 8086 receives DMA requests on $\overline{RQ/GT_0}$ and $\overline{RQ/GT_1}$ simultaneously, the DMA request on $\overline{RQ/GT_0}$ will be given priority. Thus will not be acknowledged till the DMA operation for $\overline{RQ/GT_1}$ is complete. However, if the DMA operation on $\overline{RQ/GT_0}$ is in progress, and DMA request on is received, the same will not be acknowledged till the DMA operation using $\overline{RQ/GT_1}$ ends.

Since DMA requests may occur while the 8086 is executing various instructions, it may create complications at times when the 8086 is executing instructions which read the contents of memory location, modify the contents and then write them back. Most of the arithmetic and logic instructions, with memory locations as destinations, come under this. The 16-bit memory read/write operation, with address on odd address boundary, may also get affected by DMA operations. Let us consider these with the help of some examples

(a) Instruction AND DATA, AL

This instruction will be executed in the following steps:

- (a) Memory read bus cycle to read DATA from memory.
- (b) AND operation between the contents of memory location DATA and AL register.
- (c) Memory write bus cycle to write the result in the memory location DATA.

Now suppose a DMA request is received when the first step, i.e. the memory read bus cycle is being executed. At the end of memory read

bus cycle, i.e. T₄ clock cycle, the DMA Acknowledgement will be sent and the address/data and control lines will be floated.

The second step, i.e. the AND operation, can be performed without needing any address/data and control lines. Thus, this step (b) is performed simultaneously with the DMA operation.

Suppose the AL register content is A0H and the memory location DATA content is 0BH, the result of this AND operation will then be 00H. Now let us assume that the DMA operation involves writing to a block of memory location the data of temperature at different processes in a plant. This block of memory location also includes DATA which is the temperature in the process room.

Now the external logic through DMA sends a temperature data of 70H as the temperature to be stored in DATA. After the DMA operation, the step (c) of instruction is performed, i.e. the value 00H is stored in DATA memory location. The external logic expects the data to be 70H. This may create serious errors.

(b) Instruction MOV AX, DATAW

The DATAW is a 16-bit data at location 00F07H, i.e. odd address boundary. If 00F07H and 00F08H contain 00 and FFH respectively, AX should have FF00H at the end of the operation. As explained previously, the execution involves the following steps:

- (i) Memory read cycle with 00F07H on address lines with $\overline{BHE} = 0$ and $A_0 = 1$. The data is placed in AL register or in 0 to 7 bits of the AX register.
- (ii) Memory read cycle with 00F08H on address lines with $\overline{BHE} = 1$ and $A_0 = 0$. The data is placed in AH register or in 8 to 15 bits of the AX register.

Now if the DMA request is received at the beginning of the first memory read cycle, then the DMA operation will take place between two memory read cycles. Suppose the DMA operation involves writing to location 00F08H, let us assume a byte value of 77H, then:

- The first memory read cycle will place 00 in AL.
- DMA operation will modify 00F08H to 77H.
- The second memory read cycle will place 77H in AH. Thus, AX will contain 0 to 7 bits of unmodified and 8 to 15 bits of modified values.

Both the above examples spell nightmare for the programmer as it is virtually impossible to debug the software. It is thus important that DMA operation is prevented from occurring in between the execution of an instruction. The instruction LOCK of the 8086 performs this action. Thus,

LOCK
AND DATA, AL
and LOCK
 MOV AX, DATAW

will prevent DMA operations occurring in between these instructions. However, it must also be noted that LOCK protects only the single instruction which follows it. Thus if the DMA request is received at the beginning of AND or MOV instruction, its acknowledgement will be delayed only till the execution of these instructions gets over.

5.14 HALT STATE

When HLT (Halt) instruction is executed, the 8086 enters the Halt state. In this state, the 8086 stops fetching and executing instructions. In the Halt state, no signals are floated. The Halt state can be terminated by an Interrupt request or a Reset. During the Halt state, DMA request on HOLD (minimum mode) pin or RQ/GT (maximum mode) pin gets recognized and the DMA operation takes place. However, on the completion of DMA, the 8086 returns to Halt state.

5.15 WAIT FOR TEST STATE

When the 8086 executes the WAIT instruction, it enters WAIT FOR TEST state. In this state, the 8086 generates an endless sequence of idle clock periods. During these clock periods, the Address/Data and control lines are not floated and BIU may execute memory read bus cycles to fill up the instruction object code queue.

A low signal on TEST input pin terminates this wait state. If a valid interrupt request is received in this wait state, the interrupt will be processed. But after the processing of interrupt, the 8086 will come to wait state.

The main utility of this facility is to synchronize the 8086 program with respect to an external signal, which may come from a timer circuit or an external logic, marking an event. The operation for synchronization

is simple. The 8086 program executes a WAIT instruction and then waits for the $\overline{\text{TEST}}$ input signal to restart the program.

Synchronization between the multiple Intel 8086-based systems (in multiprocessor environment) can be achieved in a similar way. Each Intel 8086-based system executes the WAIT instruction and then waits for the $\overline{\text{TEST}}$ input which is generated externally and fed to each processor.

5.16 COMPARISON BETWEEN THE 8086 AND THE 8088

Throughout the chapter, we pointed out as to how the 8088 differs from the 8086. Let us now present a consolidated picture of the comparison between these two processors.

- (a) The 8088 has the same arithmetic logic unit, the same register set, and the same instruction set as the 8086.
- (b) The 8088 has a 4-byte instruction queue in BIU as against the 6-byte instruction queue in case of the 8086.
- (c) The 8088 BIU fetches a new instruction byte to load into the queue, every time there is a one byte hole in the queue. On the other hand, the 8086 BIU fills the queue when its queue is having empty space of 2 bytes.
- (d) The 8088 like the 8086 has a 20-bit address bus. Thus the total memory addressing capacity is 1 MB.
- (e) Unlike the 8086, the 8088 has an 8-bit data bus. Thus it can read data from or write data to memory or I/O ports 8 bits at a time. To read 16-bit words from two consecutive memory locations, the 8088 always requires two memory read operations. In the case of the 8086, since it contains a 16-bit data bus, 8-bit or 16-bit memory read/write is possible as a single operation. The details about this are given in the section on external memory addressing.
- (f) Unlike the 8086, $\overline{\text{BHE}}$ is not present in the 8088. Therefore, the external memory interfaced will not have even or odd address banks. The external memory will therefore be a byte-oriented memory as in case of the 8085.

5.17 COMPATIBILITY BETWEEN THE 8088, THE 8086, THE 80186 AND THE 80286 PROCESSORS

1. Since the 8088 and the 8086 have exactly the same instruction set and almost the same architecture, the programs written for one processor will run on the other processor as well.
2. The 80188 is the improved version of the 8088, and the 80186 is the improved version of the 8086. In addition to 16-bit CPU, both the 80188 and the 80186 contain programmable peripheral devices on the same chip. The instruction sets of the 80188 and the 80186 are the supersets of the 8086. Thus in addition to the 8086 instruction set, the instruction set of the 80188 and that of the 80186 have a few more instructions. Therefore the programs written for the 8088 or the 8086 will execute on the 80188 or on the 80186 without any problem. However, the opposite is not true.
3. The 80286 was designed as the 16-bit advanced version of the 8086 for multitasking environments. It has two modes of operations, namely the real address mode and the virtual address mode. In the real address mode, the 80286 works as a fast 8086 processor. Thus most programs written for the 8086 may run on the 80286 processor. The 80286 has certain hardware features for its operation in the virtual address mode. In the virtual address mode, various users' programs are kept separate from each other, and the system programs are also protected from destruction by users' programs. These protections are necessary for any multitasking system.

5.18 CONCLUSION

The 8086 was designed with the view to using it for the development of general purpose computer systems. The earlier personal computers were based on the 8086 family (the 8088, the 80186, the 80286, etc.). Also the processor design had introduced a number of new concepts like pipelining, memory segmentation, cache memory, etc. Later microprocessors used these very concepts further to achieve more processing power.

EXERCISES

1. What are the differences between the 8086 and the 8088 microprocessors?

2. What is the maximum mode? What is its utility? What signals are output in the maximum mode as against the minimum mode?
3. What are the functions of Execution Unit and Bus Interface Unit? Explain their utility by taking examples of five instructions.
4. What is pipeline flushing? When does it occur in the 8086 system. Explain with the help of an example how pipeline flushing affects the system performance.
5. Explain the operation of the following pins.
 - (a) DT/R pin
 - (b) RESET pin
 - (c) LOCK pin
 - (d) TEST pin
 - (e) DEN pin
6. Explain the wait state generation using the READY pin. What are the timing synchronization requirements and how are they achieved? What would happen if the READY pin is grounded?
7. Draw the bus cycle timing diagram for the memory read and memory write in minimum mode. List the activities in each clock cycle.
8. (a) If a crystal of frequency 15 MHz is attached to the 8284, what would be the frequency of signals at CLK and PCLK pins?
(b) How much time does a 16-bit memory read operation at address 03B50H take? Explain.
(c) How much time does a 16-bit memory read operation at address 02A41H take? Explain.
9. Why is the 8086 memory organized into two banks of even and odd addresses? How is bank selection achieved using the BHE and A₀ signals?
10. Explain the functions of the 8288 bus controller in maximum mode configuration of the 8086.
11. Draw the basic minimum mode system configuration using the 8086 without memory or I/O interface, and list the functions performed by each chip.
12. Draw the basic maximum mode system configuration without memory or I/O interface, and list the functions performed by each chip.

13. Why have the signals $\overline{\text{AMWC}}$ and $\overline{\text{AIOWC}}$ been provided for memory and I/O write operations? Suppose a system uses these signals instead of $\overline{\text{MWTC}}$ and $\overline{\text{IOWC}}$. In that case, will there be any advantage? Explain.
14. Draw a bus cycle timing diagram for I/O read and I/O write operation in maximum mode. List the activities in each clock cycle.
15. Eight memory chips of 2 KB (2716/6116) need to be interfaced starting from 00000H. However, an address gap of 2 KB is to be left after every 4 KB of memory. Work out the address range and address decoding including $\overline{\text{G1}}, \overline{\text{G2A}}, \overline{\text{G2B}}$ for 74LS138 for the interfacing of each chip.
16. Draw a memory interface diagram in minimum mode of the 8086 for interfacing 4 KB in even and 4 KB in odd address bank, starting from address 0A000H. Explain the decoding strategy using the 74LS138 decoder.
17. Draw the maximum mode configuration of the 8086 with 2 KB memory. Take an example of memory read and list out the functions performed by each chip and the signals produced in the sequential manner.
18. Draw the timing diagram for the maskable interrupt acknowledgement cycle. List the activities in each clock cycle.
19. What will happen when the following two interrupt requests occur at the same time?
 - (i) NMI and INTO
 - (ii) NMI and INTR
 - (iii) Divide-by-Zero interrupt and NMI
 - (iv) Single Step Mode and INTR
20. In a shop floor of a factory, a foreman distributes the tasks to different work-groups (say 5) to be completed in a shift of 8 hours, using a task management system. Each work-group reports the progress of the task using a system unit connected to the main system, at the end of the shift in a simple two-state communication—completed or delayed. Plan an Intel 8086-based system in maximum mode. Draw a block diagram showing the functions of the main unit and the work-group unit.

FURTHER READING

Brey, Barry B., *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor: Architecture, Programming and Interfacing*, 4th ed., Prentice-Hall of India, New Delhi, 2001.

Intel Data book on Intel 8086, Intel Corp., Santa Clara, USA.

Osborne Adam and Kane Jerry, *Some Real Microprocessors*, Osborne Associates, Inc. Berkeley, California, 1978.

RQ/GT₀

RQ/GT₁

6

INTEL 8086 MICROPROCESSOR INSTRUCTION SET AND PROGRAMMING

6.1 INTRODUCTION

In the present chapter, we shall deal with the programming of both the 8086 and 8088 microprocessors. The programmer's model, the addressing modes as well as the instruction sets of both the microprocessors are exactly the same. The difference between these microprocessors is mainly in hardware which has been described in Chapter 5. The 8086 introduced a number of new concepts both in addressing modes and instruction sets. The instruction set of the 8086 is upward compatible with that of the 8080A at the source program level. Thus, every instruction of the 8080A can be converted to one or more of the 8086 assembly level instructions.

6.2 PROGRAMMER'S MODEL OF INTEL 8086

The programmer's model of the 8086 is shown in Figure 6.1. The 8086 has 20 address lines, i.e. total 1 MB of memory may be interfaced to it. The memory space may be divided into four segments, each of 64 KB. The segments are code segment, data segment, stack segment and extra segment. To facilitate memory addressing within the various segments, four 16-bit segment registers, namely Code Segment (CS), Data Segment (DS), Stack Segment (SS) and Extra Segment (ES) registers are provided. The segment registers store the starting address of segments in the memory. There is no restriction on the address in segment registers.

Thus, the address space in memory for two or more segments may overlap.

AX, BX, CX and DX are the four 16-bit general purpose registers. Each of these can be addressed as two 8-bit registers as shown in Figure 6.1. Thus, AH and AL may be used as 8-bit registers apart from the 16-bit register AX. Similar is the case for BX, CX and DX. These registers also have some special uses as described in the following.

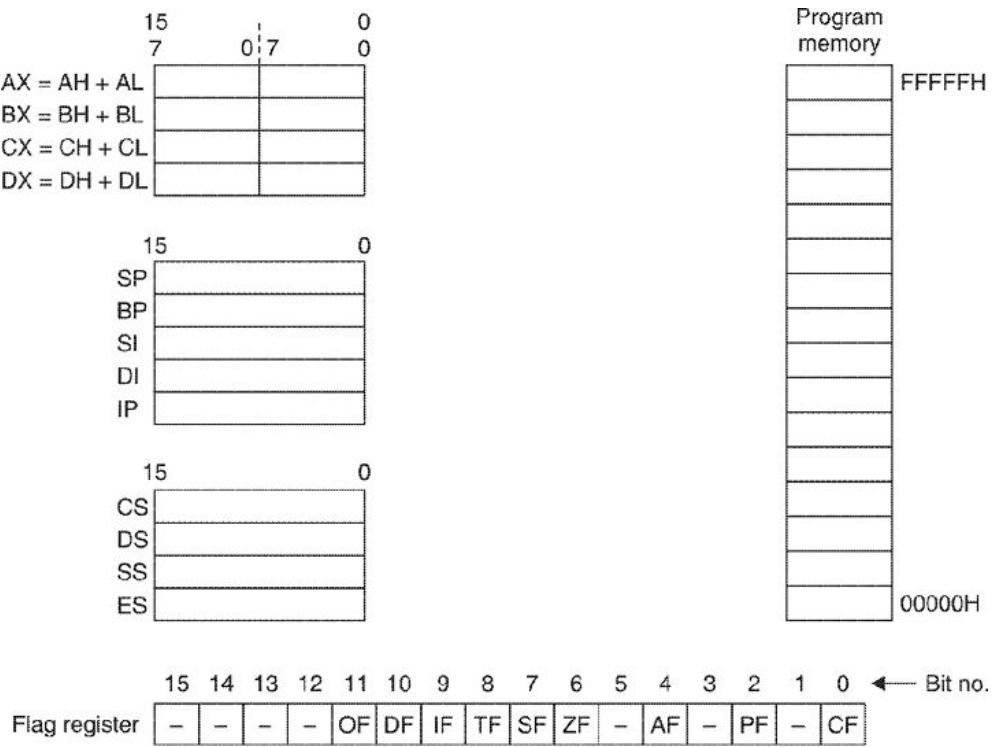


Figure 6.1 The 8086 programmer's model.

AX – Register AX serves as a primary Accumulator. It is unique in the following ways:

- Input/output operations pass data through AX (or AL).
- Instructions involving AX (or AL) and immediate data usually require less program memory than that with other registers.
- Several of powerful string primitive instructions require one of the operands to be AX (or AL).
- AX contains one-word operand and the result in 16-bit multiply and divide instructions, whereas AL is used for 8-bit operations. In 32-bit multiply and divide instructions, AX is used to hold the low-order word operand.

BX – In addition to serving as general purpose data register, BX can be used as base register when computing data memory address.

CX – In addition to serving as general purpose data register, it can be used to hold count in multi-iteration instructions. Several of the 8086 instructions can be made to repeat or loop. In such instructions, CX holds the desired number of repetitions and is automatically decremented after each iteration. When CX becomes zero the execution of instructions is terminated. Similarly, the 8-bit CL register is used as count register in bit shift and rotate instructions.

DX – In addition to being general purpose data register, DX may be used in I/O instructions, multiply and divide instructions. DX contains the address of the I/O ports in certain types of I/O instructions. In 32-bit multiply and divide instructions, DX is used to hold the high-order word operand.

Registers SI (Source Index) and DI (Destination Index) help in indexed addressing in the 8086. Similarly, the register BP (Base Pointer) is used for indirect addressing and base relative addressing. The utility of these registers will become clear when we discuss addressing modes.

The IP (Instruction Pointer) register serves as Program Counter. The IP register contents are added to the CS (Code Segment) register contents to get the Instruction address in memory. The 8086 has a 16-bit Flag Register which contains the following status flags.

- Bit 0 : Carry
- Bit 2 : Even Parity
- Bit 4 : Auxiliary Carry
- Bit 6 : Zero
- Bit 7 : Sign
- Bit 8 : Trap (used for single stepping)
- Bit 9 : Interrupt enable/disable
(0 = disable, 1 = enable)
- Bit 10 : Direction, i.e auto increment or auto decrement (of SI and DI) in string operation instruction. When DF = 1, auto decrement is effected; and when DF = 0, auto increment mode is effected.
- Bit 11 : Overflow
- Other bits : Reserved and not used.

6.3 OPERAND TYPES

The 8086 supports the following types of operands:

- Bytes
- Words
- Short Integers
- Integers
- Double words
- Long Integers
- Strings

Bytes and short integers are 8-bit variables, whereas words and integers are 16-bit variables. The double words and long integers are 32-bit variables, used as dividend in 32-bit by 16-bit divide and used as product of 16-bit by 16-bit multiply. Bytes, words and double words are unsigned numbers whereas short integers, integers and long integers represent signed numbers.

A string is a series of bytes or a series of words stored in sequential memory locations. Normally, a string is a series of alphanumeric characters defined by ASCII codes. The 8086 has a number of special instructions for string movement, string loading, string storing, string comparison, etc.

6.4 OPERAND ADDRESSING

We shall now discuss the ways in which an operand in memory may be accessed in the 8086 instructions.

The 8086 memory addresses are calculated by adding the segment register contents to an offset address. The offset address calculation depends on the addressing mode being used. The total number of address lines in the 8086 is 20 whereas the segment registers are 16 bits. The actual address in memory (effective address) is calculated as per the following steps.

- The segment register contents are multiplied by 10H, thus, shifting the contents left by 4 bits. This results in the starting address of the segment in memory.
- The offset address is calculated. The offset address is basically the offset of the actual memory location from the starting

location of the segment. The calculation of this offset value depends on the addressing mode being used.

- The offset address is added to the starting address of the segment to get the effective address, i.e. the actual memory address.

Suppose a segment register contents are xyzwH, and the offset value calculated is abcdH, then:

- Starting address of the segment = xyzwH \square 10H = xyzw0H
- Offset address = 0abcdH
- Effective address = xyzw0H + 0abcdH = ijkldH

In case of instruction addressing, the offset value is stored in IP (Instruction Pointer) register, and the code segment is used as memory segment for storing the program. As an example (Figure 6.2), if IP and CS contain 0245H and 3250H respectively, then:

- Starting address of code segment = CS contents \square 10H = 3250H \square 10H = 32500H
- Offset address = contents of IP = 0245H
- Effective address of instruction = 32500H + 0245H = 32745H

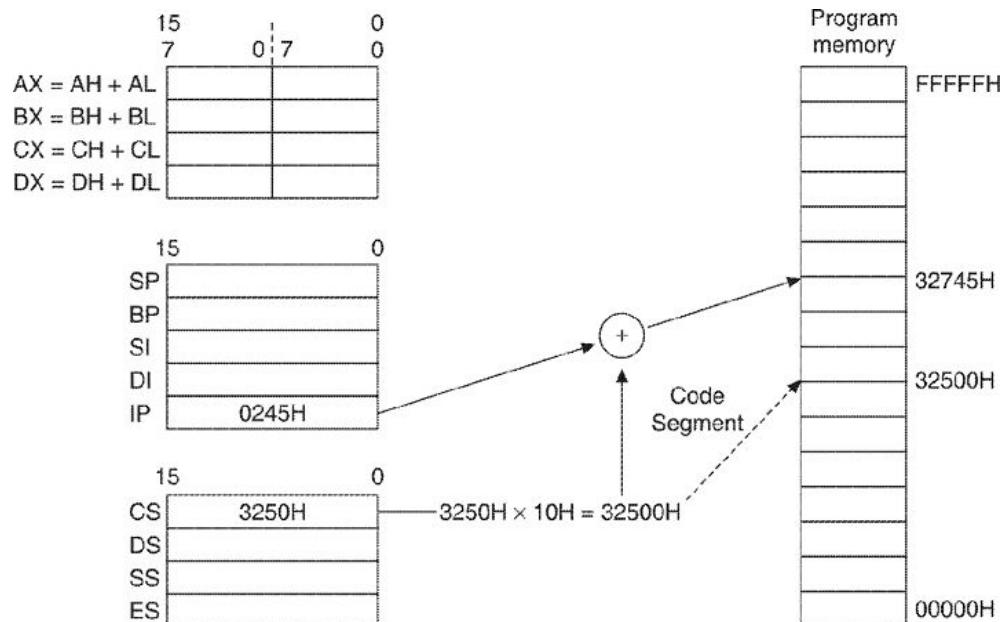


Figure 6.2 Instruction addressing.

The addressing modes clearly specify the location of the operand and also how its location may be determined. The following addressing

modes are supported in the 8086.

- Register Addressing Mode
- Immediate Addressing Mode
- Direct Memory Addressing Mode
- Register Indirect Addressing Mode
- Base plus Index Register Addressing Mode
- Register Relative Addressing Mode
- Base plus Index Register Relative Addressing Mode
- String Addressing Mode

6.4.1 Register Addressing Mode

When both destination and source operands reside in registers, the addressing mode is known as register addressing mode. Following are the examples:

1. **MOV AX, BX**
 $(AX) \square (BX)$
Move the contents of BX register to AX register. The contents of BX register remain unchanged.
2. **AND AL, BL**
 $(AL) \square (AL) \text{ AND } (BL)$
AND the contents of AL register with the contents of BL register and place the resultant contents in AL register.

6.4.2 Immediate Addressing Mode

When one of the operands is part of the instruction, the addressing mode is known as immediate addressing mode. The operand (8/16 bits) immediately follows the instruction operation code. Some representative examples are given below.

1. **MOV CX, 2346H**
 $(CX) \square 2346H$
Copy into CX the 16-bit data 2346H
2. **SUB AL, 24H**
 $(AL) \square (AL) - 24H$
Subtract 24H from the contents of AL register and put the result in AL register.

6.4.3 Direct Memory Addressing Mode

In this mode, the 16-bit offset address is part of the instruction as displacement field. It is stored as 16-bit unsigned or 8-bit sign-extended number, immediately following the instruction opcode. Some examples are given below.

1. MOV (4625H), DX

Copy the contents of DX register into memory locations calculated from Data Segment register and offset 4625H.

$((DS) \square 10H + 4625H) \square (DL)$

$((DS) \square 10H + 4625H + 1) \square (DH)$

The lower byte (DL) is stored in the lower memory location, and the higher byte (DH) is stored in the higher memory location.

2. OR AL, (3030H)

OR the contents of AL register with the contents of memory location calculated from DS register and offset 3030H. The result is copied into AL register.

$(AL) \square ((DS) \square 10H + 3030H) OR (AL)$

If AL contains 70H before instruction execution and DS contains 3745H, then the memory location of operand = $(DS) \square 10H + 3030H = 37450H + 3030H = 3A480H$

If the memory location 3A480H has 57H as contents, then

$(AL) \square 57H OR 70H \square 77H$

The operation is explained in Figure 6.3.

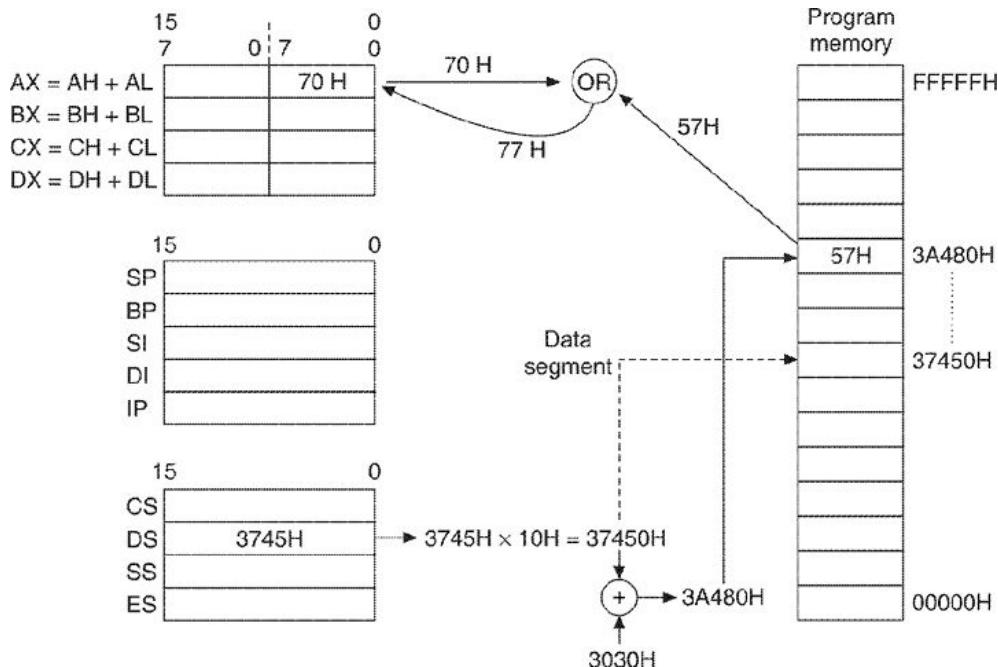


Figure 6.3 Execution of direct memory addressed instruction `OR AL, (3030H)`.

6.4.4 Register Indirect Addressing Mode

In this addressing mode, the offset address is specified through pointer register or index register.

For index register, the SI (Source Index) register or DI (Destination Index) register may be used, whereas for pointer register, BX (Base Register) register or BP (Base Pointer) register may be used. Following are some examples of the application of the register indirect addressing mode.

1. `MOV AL, (BP)`

Copy into AL register the contents of memory location, whose address is calculated using offset as contents of BP register and the contents of DS register.

$$(AL) \square ((DS) \square 10H + (BP))$$

2. `XOR (DI), CL`

Perform XOR operation between the contents of CL register and the contents of memory location whose address is calculated using the offset stored in DI register and the contents of the DS register. The result is stored in the memory location.

$$((DS) \square 10H + (DI)) \square ((DS) \square 10H + (DI)) \text{ XOR } (CL)$$

If DS contains `1400H`, DI contains `0056H` and CL contains `7FH`, then

$$(DS) \square 10H + (DI) = 14000H + 0056H = 14056H$$

If the memory location 14056H contains 43H, then
 $43H \text{ XOR } 7FH = 3CH$
 3CH is stored in memory location 14056H. The operation is illustrated in Figure 6.4.

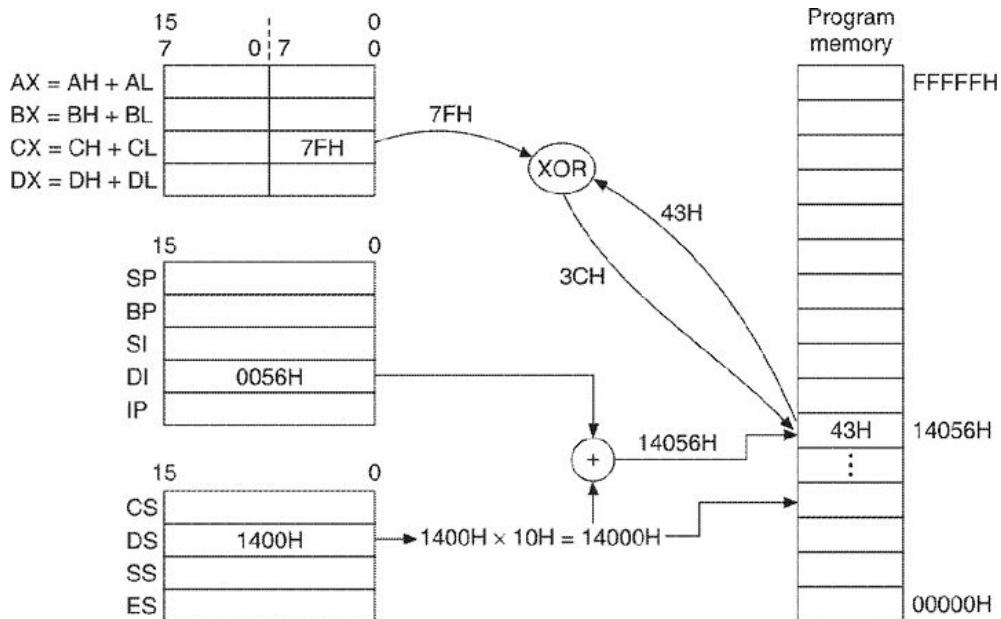


Figure 6.4 Execution of register indirect addressing mode instruction `XOR (DI), CL`.

6.4.5 Base Plus Index Register Addressing Mode

The main utility of this addressing mode is when an array of data is to be addressed. It is the extension of register indirect addressing mode. In this mode, both base register (BP or BX) and index register (SI or DI) are used to indirectly address the memory location. An example is given below.

`MOV (BX + DI), AL`

Copy the contents of AL register into memory location whose address is calculated using the contents of DS (Data Segment), BX (Base Register) and DI (Destination Index) registers.

$((DS) \square 10H + (BX) + (DI)) \square (AL)$

The base register BX may contain the offset to address the location of the beginning of the array. The value of DI may vary depending on the element of the array to be addressed. For 0th, 1st, 2nd, ..., etc. elements DI will have values 0, 1, 2, ..., and so on.

If DS contains 1000H, BX contains 0500H, DI contains 0 and AL contains 24H, we get

$$(DS) \square 10H + (BX) = 1000H + 0500H = 10500H$$

Since DI contains 0, the actual memory location address, i.e. effective address = 10500H.

Thus 24H is copied to memory location 10500H. If DI was made 1, 24H will be copied to the next location, i.e. 10501H, by using the same instruction. Therefore, any operation on a number of elements of the array may be performed by incrementing/decrementing DI and putting the operation in a loop. The operation is illustrated in Figure 6.5.

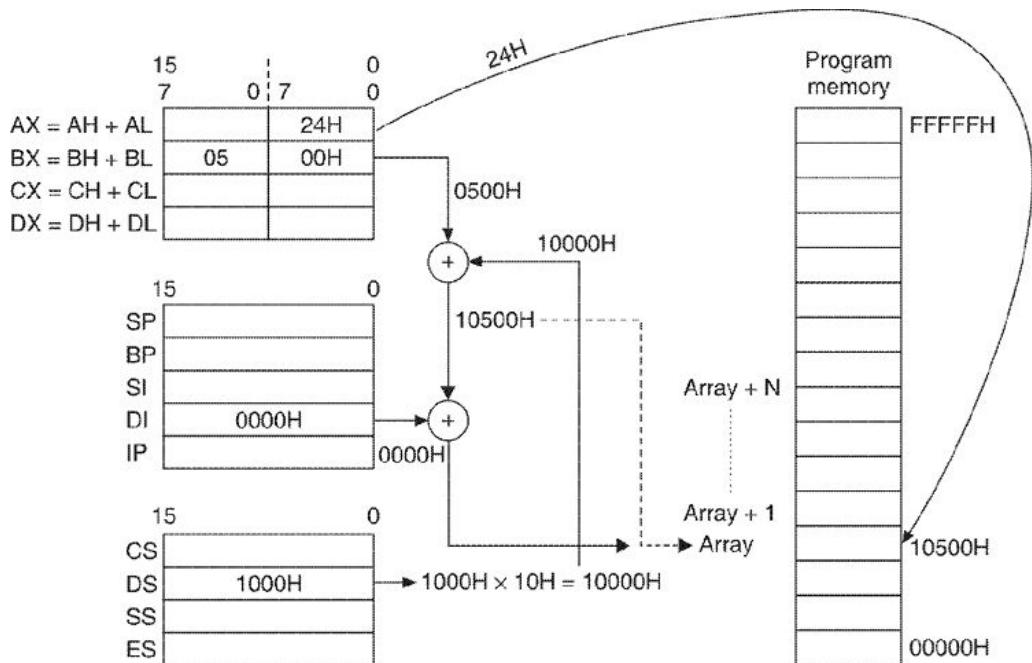


Figure 6.5 Execution of base plus index register addressing mode instruction $\text{MOV} (\text{BX} + \text{DI}), \text{AL}$.

6.4.6 Register Relative Addressing Mode

This mode is similar to base plus index addressing mode. In this mode, the offset is calculated using either a base register (BP, BX) or an index register (SI, DI) and the displacement specified as an 8-bit or a 16-bit number, as part of the instruction. The displacement is specified for addition or subtraction like $\text{BX} + 3$, $\text{DI} - 0050\text{H}$, etc.

Like the base plus index addressing mode, the register relative addressing mode can also be used to access different elements of the array. The base register or index register may point to either the beginning or the end of the array. Displacement indicates the distance of the array element to be accessed from the beginning or end of the array of elements. Let us take the following example to understand it better.

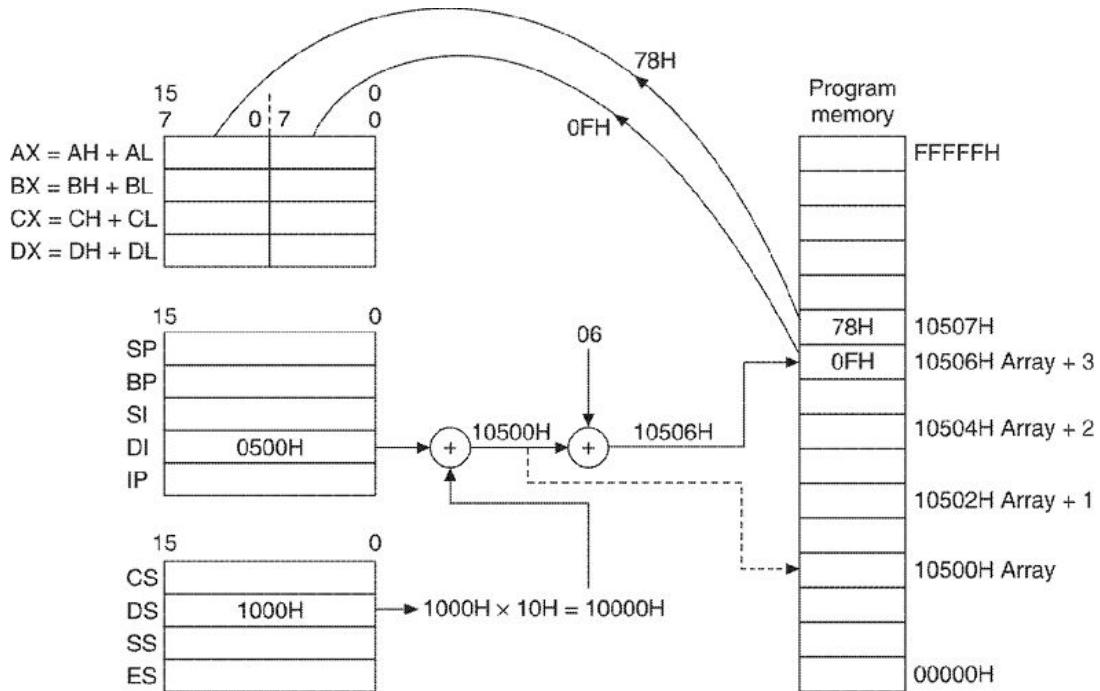


Figure 6.6 Execution of register relative addressing mode instruction `MOV AX, (DI + 06)`.

`MOV AX, (DI + 06)`

Copy to AL the contents of memory location whose address is calculated using DS (Data Segment), DI (Destination Index) register with displacement of 06, and copy to AH the contents of the next higher memory location.

$$(AL) \square ((DS) \square 10H + (DI) + 06)$$

$$(AH) \square ((DS) \square 10H + (DI) + 07)$$

These data bytes may be part of the 16-bit array. By increasing/decreasing the displacement, we may access different elements of the array. If DS contains `1000H` and DI contains `0500H`, we get

$$(DS) \square 10H = 10000H$$

$$(DS) \square 10H + (DI) = 10500H$$

$$(DS) \square 10H + (DI) + 06H = 10506H$$

Effective address of the lower byte = `10506H`

Effective address of the higher byte = `10507H`

If the contents at lower and higher memory locations are `0FH` and `78H` respectively, the AL will contain `0FH` and the AH will contain `78H` and thus AX will have `780FH` as 16-bit value as a result of the execution of this instruction. Figure 6.6 illustrates the operation of the instruction.

6.4.7 Base Plus Index Register Relative Addressing Mode

This addressing mode is basically the combination of base plus index register addressing mode and register relative addressing mode. To find the address of the operand in memory, a base register (BP or BX), an index register (DI or SI) and the displacement which is specified in instruction is used along with the data segment register. For example:

`MOV (BX + DI + 2), CL`

Copy the contents of the CL register to the memory location whose address is calculated using DS (Data Segment), BX (Base Register) and DI (Destination Index) registers and 02 as displacement.

$$((DS) \square 10H + (BX) + (DI) + 2) \square (CL)$$

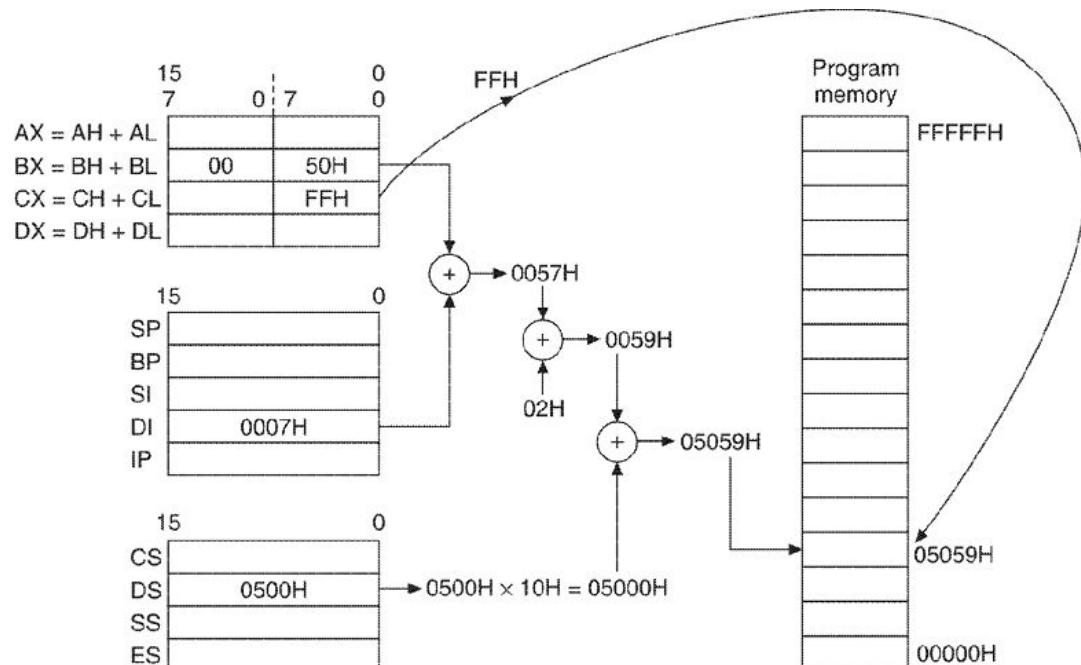


Figure 6.7 Execution of base plus index register relative addressing mode instruction `MOV (BX + DI + 2), CL`.

If DS contains 0500H, BX contains 0050H, DI contains 0007H, we get

$$(DS) \square 10H = 05000H$$

$$\begin{aligned} (BX) + (DI) + 2 &= 0050H + 0007H + 02H \\ &= 0059H \end{aligned}$$

$$((DS) \square 10H + (BX) + (DI) + 2) = 05059H$$

If CL contains FFH, this value will be copied to memory location 05059H. Figure 6.7 illustrates the operation.

You may wonder about the utility of this addressing mode which is the combination of two addressing modes. Actually this addressing mode is very useful while accessing two-dimensional arrays. Consider the following two-dimensional array:

X00	X01	X02	X03
X10	X11	X12	X13
—	—	—	—
—	—	—	—
—	—	—	—

The elements will be stored in memory sequentially like X00, X01, X02, X03, X10, X11, X12, X13, and so on. Each row may represent elements of the record of an employee in a factory like Code No., Age, Gender, No. of Days worked, etc. The applications may be the calculation of salaries, maintaining the record of employees with respect to age, gender, etc. To access any information about an employee, it will be required to

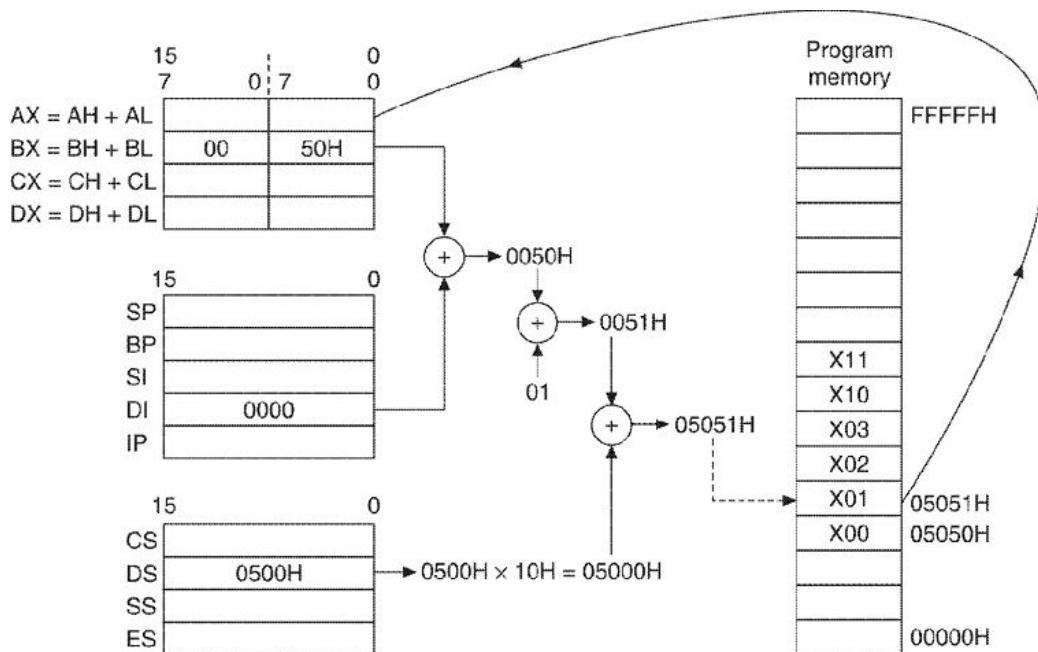


Figure 6.8 Accessing a two-dimensional array using base plus index register relative addressing mode instruction `MOV AL, (BX + DI + 1)`.

- locate the starting of array,
- locate the employees record (i.e. the row of the array), and
- locate and access the element containing the information in the record (i.e. the column number in the row)

Using base pointer, we may locate the array starting location in the memory. Using base pointer and index register, we may locate the particular row. Using the combination of base pointer, index register and displacement, we may locate and access the particular column. Figure 6.8 illustrates the accessing of a two-dimensional array using base plus index register relative addressing mode when DS = 0500H, BX = 0050H, DI = 0000H.

The instruction execution illustrated is MOV AL, (BX + DI + 1), i.e. X01 will be copied to AL register. To access different records (i.e. rows), DI can be increased/decreased by 4 as the case may be. To access different elements within a row, the displacement value may be specified appropriately.

6.4.8 String Addressing Mode

The accessing of the string operands is different from that of the other operands. There are special instructions for string operations. Apart from the segment registers, the index registers SI and DI are used. The segment register DS and the index register SI are used for source string, whereas for destination string, the segment register ES and the index register DI are used. The direction flag bit DF in flag register plays an important role in string instructions. If DF = 0, both SI and DI are incremented and if DF = 1, both SI and DI are decremented automatically as part of the string instruction execution. Thus SI and DI point to the next byte or word of the string. This helps in executing the instruction in loop. DF can be cleared (made 0) by CLD instruction and can be set (made 1) by STD instruction. An example is given below.

MOVSB

Copy the byte from the source string location determined by DS and SI to the destination string location determined by ES and DI.

If (DS) = 0500H, (SI) = 0000H,

(ES) = 0F00H, (DI) = 0000H and (DF) = 0

$$\text{Source location} = (\text{DS}) \square 10H + (\text{SI})$$

$$= 05000H + 0000 = 05000H$$

$$\text{Destination location} = (\text{ES}) \square 10H + (\text{DI})$$

$$= 0F000H + 0000 = 0F000H$$

If 96H is contained in location 05000H, it will be copied to location 0F000H. Both SI and DI will be incremented by 1.

Figure 6.9 illustrates the operation to this instruction.

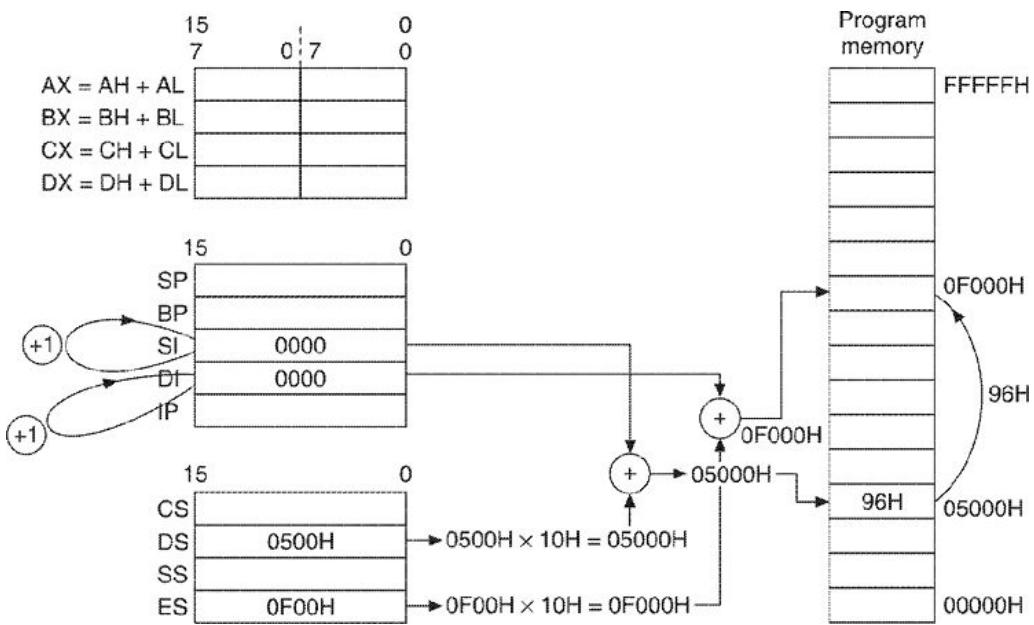


Figure 6.9 Execution of string addressing mode instruction, MOVSB.

6.5 THE 8086 ASSEMBLER DIRECTIVES

We discussed the utility of the assembler directives in Chapter 2. Some directives such as ORG, EQU, DB, DW, etc. which are common in different assemblers were also described. Here, we shall deal with the assembler directives that are specific to the 8086 assembly language. Though a representative set of directives for the 8086 assembler is presented, it is possible that some assemblers have a few additional directives. On the other hand, some of the directives presented here may not be present or may be present in different forms.

6.5.1 Directives for Constant and Variable Definition

An Intel 8086 assembly program uses different types of constants like binary, decimal, octal and hexadecimal. These can be represented in the program using different suffixes like B (for binary), D (for decimal), O (for octal) and H (for hexadecimal) to the constant. For example:

- 10H is a hexadecimal number (equivalent to decimal 16).
- 27O is an octal number (equivalent to decimal 23).
- 10100B is a binary number.

A number of directives are used to define and store different kinds of constants.

The DB (Define Byte), DW (Define Word), DDW (Define Double Word) are described in Chapter 2. The 8086 assembler uses DD as directive for double word. Some of the 8086 assemblers also provide the following additional directives.

DQ : Define Quadword

DT : Define Ten Bytes

In addition, these directives can be used to store strings and arrays. For example:

- NAME DB “NEHA SHIKHA”. The ASCII codes of alphanumeric characters specified within double quotes are stored.
- NUM DB 5, 6, 10, 12, 7. The array of five numbers NUM is declared and the numbers 5, 6, 10, 12 and 7 are stored.

The directives DUP and EQU are used to store strings and arrays.

DUP

Using the DUP directive, several locations may be initialized and the values may be put in those locations. The format is as follows.

Name Type Num DUP (value)

For example: TEMP DB 20 DUP (5)

The above directive defines an array of 20 bytes in memory and each location is initialized to 5. The array is named TEMP.

EQU

As mentioned in Chapter 2, the EQU directive may be used to define a data name with immediate value or another data name. It can also be used to equate a name to a string. For example:

NUMB EQU 20H

NAME EQU “RASHMI”

6.5.2 Program Location Control Directives

The directives used for program location control in the 8086 assembler are ORG, EVEN, ALIGN and LABEL.

ORG

The ORG directive is used to set the location counter to a particular value. For example:

ORG 2375H

The location counter is set to 2375H. If it occurs in data segment, the next data storage will start at 2375H. If it is in code segment, the next instruction will start at 2375H.

EVEN

Using EVEN directive, the next data item or label is made to start at the even address boundary. The assembler, on encountering EVEN directive, will advance the location counter to even address boundary. For example:

EVEN TABLE DW 20 DUP (0)

This statement declares an array named TABLE of 20 words starting from the even address. Each word is initialized to zero.

ALIGN number

This directive will force the assembler to align the next segment to an address that is divisible by 2, 4, 8 or 16. The unused bytes are filled with 0 for data and NOP instruction for code. For example:

ALIGN 2

It will force the next segment to the next even address.

(*Note:* Normally ALIGN 2 directive is used to force the data segment on word boundary and ALIGN 4 is used to force the data segment on double word boundary.)

LENGTH

It is an operator which is used to tell the assembler to determine the number of elements in a data item, such as string or array. For example:

MOV CX, LENGTH ARRAY

This statement will move the length of the array to the CX register.

OFFSET

This operator is used to determine through the assembler, the offset of a data item from the start of the segment containing it. For example:

MOV AX, OFFSET FACT

This statement will place in the AX register the offset of the variable FACT from the start of the segment.

LABEL

The LABEL directive is used to assign a name to the current value in the location counter. The location counter is used by the assembler to keep track of the current location. The value in the location counter denotes the distance of the current location from the start of the segment. The LABEL directive may be used to specify the destination of the branch-

related instruction using jump or call instructions. It may also be used to specify a data item. When the LABEL directive is used to specify the destination, it is necessary to specify the label as NEAR or FAR. When the destination of jump/call is in the same segment, the label is specified as NEAR; otherwise when the destination is in another segment, it is specified as FAR. For example:

TERM LABEL FAR

The TERM is the label and to this point jump from another segment may be effected. It must be noted that without declaring the label as FAR, it is not possible to perform the jump to the label from another segment. When the label is used to specify a data item, the type of the data item must be specified. The data may be type byte, or type word. For example:

```
SSEG SEGMENT STACK  
DW 50 DUP(0)  
STACK-TOP LABEL WORD  
SSEG ENDS
```

The first statement declares a SSEG as name of the stack segment. The second statement reserves 50 words and fills them with 0. The next statement assigns the name STACK-TOP to the word above the stack top.

6.5.3 Segment Declaration Directives

These directives help in declaring various segments with specific names. The start and the end of segments may also be specified using these directives. The directives for segment declaration include SEGMENT, ENDS, ASSUME, GROUP, CODE, DATA, STACK etc.

SEGMENT and ENDS

The SEGMENT and ENDS directives signify the start and end of a segment. Data items or instructions placed between SEGMENT and ENDS directives identify a group with the name assigned to segment. SEGMENT and ENDS directives are used in the same way as parentheses are used in group in algebra. For example:

```
INST SEGMENT  
ASSUME CS: INST, DS: DATAW  
—  
—  
—
```

INST ENDS

ASSUME

This directive is used to assign logical segment to physical segment at any time. It tells the assembler as to what addresses will be in segment registers at the time of execution.

For example:

```
ASSUME CS: CODE, DS: DATA, SS: STACK
```

This directive tells the assembler that the CS register will store the address of the segment whose name is CODE, and so on.

.CODE (Name)

This code directive is the shortcut used in the definition of code segments. The name is optional and is specified if there is more than one code segment in the program.

.DATA and .STACK

Similar to .CODE, the .DATA and .STACK directives are shortcuts in the definition of data segment and stack segment, respectively.

GROUP

This directive is used to tell the assembler to group all the segments in one logical group segment. This allows the contents of all the segments to be accessed from the same segment base. For example:

```
PROG GROUP CODE, DATA
```

The above statement will group the two segments CODE and DATA into one segment named PROG. Each segment must be declared using ASSUME statement as in the following statement.

```
ASSUME CS: PROG, DS: PROG
```

6.5.4 Procedure and Macro-related Directives

The directives in this group relate to the declaration of procedures and macros along with the variables contained in them. The directives include PROC, ENDP, PUBLIC, MACRO, ENDM and EXTRN.

PROC and ENDP

The PROC directive is used to define the procedures. The procedure name is a *must* and it must follow the naming convention of the assembler. Along with the name of the procedure, the field NEAR or FAR also needs to be specified. If the procedure is resident in the same segment where it is called, it is said to be of type NEAR. On the other

hand, if the procedure is resident in a segment which is different from the segment where it is called, it is said to be of type FAR.

The ENDP directive is used to mark the end of the procedure. Some examples are given below.

```
 FUNCT PROC FAR
```

```
—  
—
```

```
ENDP
```

```
 FACT PROC NEAR
```

```
—  
—
```

```
ENDP
```

The first procedure FUNCT is in a segment which is different from where it is called. The second procedure FACT is in the same segment where it is called. All the statements of the procedure are between PROC and ENDP directives.

PUBLIC

Normally the programs which are written to solve real life problems, are complex and such programs are developed in separate modules. The programs follow a well-laid hierarchy as described in Chapter 1. The modules are developed separately and later their object code files are linked to form a complete program. It is very much possible that a variable is defined in one module, but is used in other modules. In order to facilitate the linking, such variables are declared *public*, using the PUBLIC directive in the module where they are defined. For example:

```
 PUBLIC PX, PY, PZ
```

The above statement in a module will make the variables PX, PY and PZ available to other modules.

MACRO and ENDM

The MACRO directive is used to define macros in the program. The ENDM directive defines the end of MACRO.

EXTRN

This directive is used in case the procedure called in the program is not part of it. The EXTRN directive conveys to the assembler that the names and labels mentioned are part of some other module, i.e. external to the module being assembled. It is necessary that the names and labels

mentioned in EXTRN are also declared public. The assembler along with the linker, takes the necessary action to link the modules.

6.5.5 Other Directives

These directives are of general nature, or they relate to more than one group described earlier. The directives described include PTR, PAGE, TITLE, NAME and END.

PTR

The PTR is an operator used in instructions to assign a specific type to a variable or a label. In a number of instructions, the type of the operand is not clear. The PTR operator is used to assign a type in such cases. As an example, the INC (BX) does not indicate whether the byte or the word in the memory location, pointed by BX, is to be incremented. The instruction INC BYTE PTR (BX) clearly tells the assembler that the byte pointed to by BX is to be incremented. The PTR operator can also be used to override the declared type of variable. Following is an example of the use of PTR directive.

```
ARRAY DW 0125H, 1630H, 9275H ...
```

In the above array of words, suppose we wish to move a byte from the array, we may then simply insert the PTR operator as follows

```
MOV AL, BYTE PTR ARRAY
```

PAGE

This directive is used for listing of the assembled program. At the start of the program, the directive is placed to specify the maximum number of lines on a page and the maximum number of characters in a line to be placed for listing. An example is given below.

```
PAGE 60, 120
```

The above example specifies that 60 lines are to be listed on a page with a maximum of 120 characters in each line.

TITLE

This directive is also used for the listing of the program. The title declared in this directive defines the title of the program and is listed in line 2 of each page of program listing. The maximum number of characters allowed is 60. For example:

```
TITLE PROGRAM TO FIND SQUARE ROOT
```

NAME

The NAME directive is used to assign a specific name to each module, when the program consists of a number of modules. This helps in

understanding the program logic.

END

This is the last statement of the program and it specifies the end of the program to the assembler.

It must be noted that not all of the above directives are used in all programs. User may deploy them depending on the need of the program logic.

6.6 INSTRUCTION SET

Figure 6.10 shows the format of the 8086 instructions. An instruction length may range between 1 and 6 bytes. The instruction will contain the codes for the operation to be performed, the addressing modes, the registers involved, the displacement and the immediate data as required. The 8086 allows both bytes and word operands in memory. Thus, both byte and word access in memory are possible. The 8086 instructions need not be word aligned.

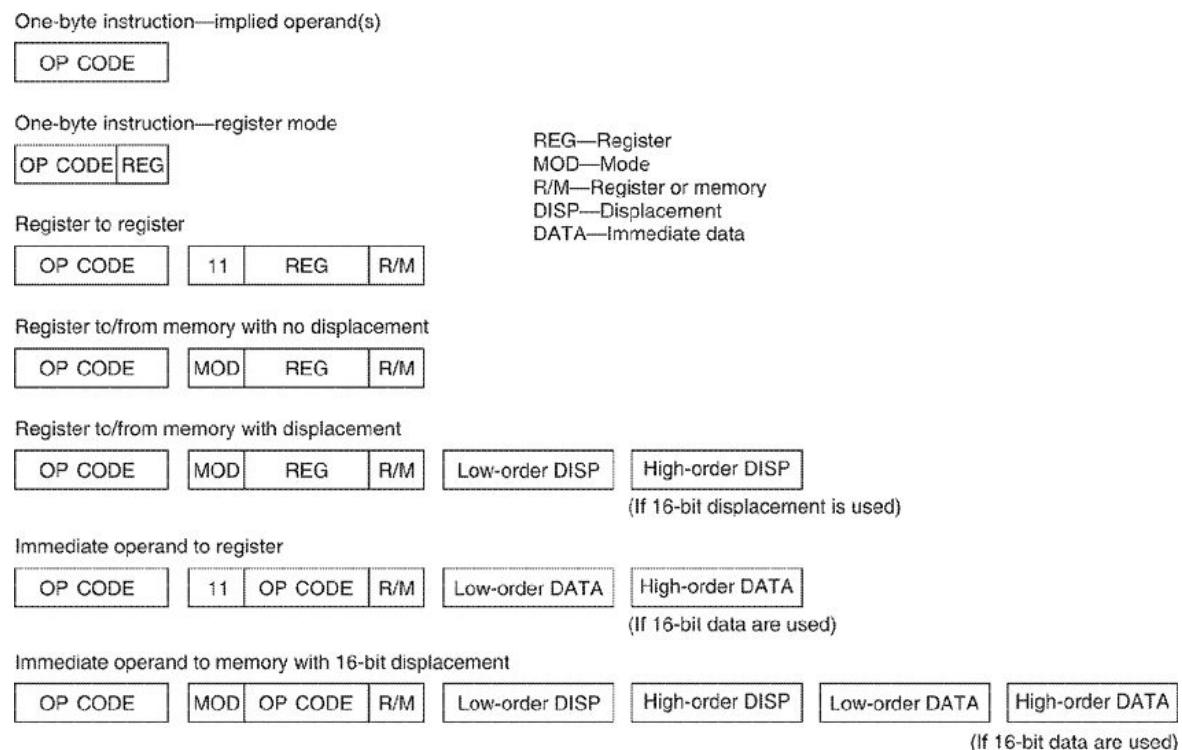


Figure 6.10 Examples of 8086 instruction formats.

We shall now discuss the various groups of instructions. Following are some of the symbols and abbreviations used.

reg : Register (8 bit or 16 bit)

reg 8, RB : Byte register

reg 16, RW : Word register
 mem. : 8- or 16-bit value in memory location defined by EA (Effective Address). The EA is calculated based on addressing mode specified, as described previously.
 mem 8 : Byte value in memory
 mem 16 : Word value in memory
 immed : Immediate data (8 bit or 16 bit)
 immed 8 : Immediate data (8 bit)
 immed 16 : Immediate data (16 bit)
 seg reg, SR : Segment register
 SFR : Status Flag register
 accum : Accumulator (AL or AX)
 Src. : Source operand
 Dest. : Destination operand
 OEA : 16-bit displacement which, when added to the segment register, creates the actual data memory address, i.e. effective address.
 DS: SI : Memory address calculated by Data Segment (DS) register contents and Source Index (SI) register contents.
 ES:DI : Memory address calculated by Extra Segment (ES) register contents and Destination Index (DI) register contents.
 PDX : Port address stored in DX register.
 EA : Effective address, i.e. actual memory address.

6.7 DATA TRANSFER GROUP

The instructions in this group perform data movement between registers, register and memory, register and immediate data, memory and immediate data, between two memory locations, between I/O port and register, and between stack and memory or register. Both 8- and 16-bit data transfers are provided. For example:

MOV dest, src—Move Byte or Word

Flags affected: None

Copies the byte or word from the source operand to the destination operand.

Register to register operation

MOV RBD, RBS (2 bytes)
(RBD) □ (RBS)

MOV RWD, RWS (2 bytes)
(RWD) □ (RWS)

Memory to register operation

MOV RB, DADDR (2, 3 or 4 bytes)
(RB) □ (EA)

MOV RW, DADDR (2, 3 or 4 bytes)
(RW) □ (EA)

Register to memory operation

MOV DADDR, RB (2, 3 or 4 bytes)
(EA) □ (RB)

MOV DADDR, RW (2, 3 or 4 bytes)
(EA) □ (RW)

Immediate data to register operation

MOV RB, DATA8 (2 bytes)
(RB) □ DATA8

MOV RW, DATA16 (3 bytes)
(RW) □ DATA16

Immediate data to memory operation

MOV DADDR, DATA8 (3, 4 or 5 bytes)
(EA) □ DATA8

MOV DADDR, DATA16 (4, 5 or 6 bytes)
(EA) □ DATA16

Register to segment register operation

MOV SR, RW (2 bytes)
(SR) □ (RW)

Segment register to register operation

MOV RW, SR (2 bytes)
(RW) □ (SR)

Memory to segment register operation

MOV SR, DADDR (2, 3 or 4 bytes)
(SR) □ (EA)

Segment register to memory operation

MOV DADDR, SR (2, 3 or 4 bytes)
(EA) □ (SR)

Memory to accumulator operation (direct addressing)

MOV AL, LABEL (3 bytes)
(AL) □ (EA)

MOV AX, LABEL (3 bytes)
(AX) □ (EA)

Accumulator to memory operation (direct addressing)

MOV LABEL, AL (3 bytes)
(EA) □ (AL)

MOV LABEL, AX (3 bytes)
(EA) □ (AX)

POP dest—Pop Word off Stack

Flags affected: None

Transfers the word at the current stack top (SS:SP) to the destination and then increments SP by 2 to point to the new stack top. CS is not a valid destination.

Stack to memory operation

POP DADDR (2, 3 or 4 bytes)
(EA) □ ((SP)), (SP) □ (SP) + 2

Stack to register operation

POP RW (1 or 2 bytes)/POP SR (1 byte)
(RW or SR) □ ((SP)), (SP) □ (SP) + 2

POPF (1 byte)—Pop Flags off Stack

Flags affected: All flags.

Pops word from stack into the Flags Register and then increments SP by 2. For example:

(SFR) □ ((SP)), (SP) □ (SP) + 2

PUSH src—Push Word onto Stack

Flags affected: None

Decrements SP by 2 and transfers one word from source to the Stack top (SS:SP).

Register to stack operation

PUSH RW (1 byte)/PUSH SR (1 byte)
(SP) □ (SP) - 2, ((SP)) □ (RW or SR)

Memory to stack operation

PUSH DADDR (2, 3 or 4 bytes)

(SP) □ (SP) – 2, ((SP)) □ (EA)

PUSHF (1 byte)—Push Flags onto Stack

Flags affected: None.

Transfers the Flags Register onto the stack. For example:

(SP) □ (SP) – 2, ((SP)) □ (SFR)

LDS reg16, mem—Load Pointer using DS

Flags affected: None.

Loads the 16-bit pointer from memory source to destination register and DS. The offset is placed in the destination register and the segment is placed in DS. To use this instruction, the word at the lower memory address must contain the offset and the word at the higher address must contain the segment. This simplifies the loading of FAR pointers from the stack and the interrupt vector table. For example:

LDS RW, DADDR (2, 3 or 4 bytes)

(RW) □ (EA), (DS) □ (EA) + 2

LEA reg, mem—Load Effective Address

Flags affected: None.

Transfers the offset address of “src” to the destination register. For example:

LEA RW, DADDR (2, 3 or 4 bytes)

(RW) □ OEA

Load into RW the 16-bit address displacement which, when added to the segment register contents, creates the effective data memory address.

LES reg16, mem—Load Pointer using ES

Flags affected: None.

Loads the 16-bit pointer from memory source to destination register and ES. The offset is placed in the destination register and the segment is placed in ES. To use this instruction, the word at the lower memory address must contain the offset and the word at the higher address must contain the segment. This simplifies the loading of the FAR pointers from the stack and the interrupt vector table. For example:

LES RW, DADDR (2, 3 or 4 bytes)

(RW) □ (EA), (ES) □ (EA) + 2

LODSB/LODSW—Load String (Byte or Word)

Flags affected: None.

Transfers the string element addressed by DS:SI to the Accumulator. SI is incremented or decremented based on the size of the operand and state

of the Direction Flag. If the Direction Flag is set, SI is decremented and if the Direction Flag is clear, SI is incremented. Use with REP prefixes. For example:

LODSB (1 byte)

(AL) \square (DS:SI), (SI) \square (SI) + 1 if (DF) = 0, (SI) \square (SI) - 1 if (DF) = 1

LODSW (1 byte)

(AX) \square (DS:SI), (SI) \square (SI) + 2 if (DF) = 0, (SI) \square (SI) - 2 if (DF) = 1

STOSB/STOSW—Store String (Byte or Word)

Flags affected: None.

Stores value in the Accumulator to the location at ES: DI. DI is incremented/decremented based on the size of the operand (or instruction format) and the state of the Direction Flag. Use with REP prefixes. For example:

STOSB (1 byte)

(ES:DI) \square (AL), (DI) \square (DI) + 1 if (DF) = 0, (DI) \square (DI) - 1 if (DF) = 1

STOSW (1 byte)

(ES:DI) \square (AX), (DI) \square (DI) + 2 if (DF) = 0, (DI) \square (DI) - 2 if (DF) = 1

MOVSB/MOVSW—Move String (Byte or Word)

Flags affected: None.

Copies data from memory location addressed by DS:SI to the location ES:DI destination and updates SI and DI based on the size of the operand and state of the Direction Flag. SI and DI are incremented when the Direction Flag is clear and decremented when the Direction Flag is set. Use with REP prefixes. For example:

MOVSB (1 byte)

(ES:DI) \square (DS:SI), (SI) \square (SI) + 1, (DI) \square (DI) + 1 if (DF) = 0

(SI) \square (SI) - 1, (DI) \square (DI) - 1 if (DF) = 1

MOVSW (1 byte)

(ES:DI) \square (DS:SI), (SI) \square (SI) + 2, (DI) \square (DI) + 2 if (DF) = 0

(SI) \square (SI) - 2, (DI) \square (DI) - 2 if (DF) = 1

REP—Repeat String Operation

Flags affected: None.

Repeats the execution of string instructions while CX > 0. After each string operation, CX is decremented and the Zero Flag is tested.

REPE/REPZ—Repeat Equal/Repeat Zero

Flags affected: None.

Repeats the execution of string instructions while CX > 0 and the Zero Flag is set. CX is decremented and the Zero Flag is tested after each string operation.

REPNE/REPNZ—Repeat Not Equal/Repeat Not Zero

Flags affected: None.

Repeats the execution of the string instructions while CX > 0 and the Zero Flag is clear. CX is decremented and the Zero Flag is tested after each string operation.

LAHF—Load Register AH from Flags

Flags affected: None.

Copies bits 0–7 of the Flag Register into AH. This includes flags AF, CF, PF, SF and ZF. Other bits are undefined. For example:

AH = SF ZF xx AF xx PF xx CF

SAHF—Store AH Register into Flags

Flags affected: AF, CF, PF, SF, ZF.

Transfers bits 0–7 of AH into the Flag Register. This includes AF, CF, PF, SF and ZF.

XCHG dest, src—Exchange

Flags affected: None.

Exchanges the contents of source and destination.

Register and register operation

XCHG RB, RB (2 bytes)

(RB) □□ (RB)

XCHG RW, RW (2 bytes)

(RW) □□ (RW)

Memory and register operation

XCHG RB, DADDR (2, 3 or 4 bytes)

(RB) □□ (EA)

XCHG RW, DADDR (2, 3 or 4 bytes)

(RW) □□ (EA)

Accumulator and register operation

XCHG AX, RW (1 bytes)

(AX) □□ (RW)

XLAT(1 byte)—Translate

Flags affected: None.

Replaces the byte in AL with the byte from a user table addressed by BX. The original value of AL is the index into the translate table.

(AL) \square ((AL) + (BX))

IN AL/AX, PORT—Input Byte or Word from Port

Flags affected: None.

A byte or word is read from “port” and placed in AL or AX respectively. If the port number is in the range of 0–255, it can be specified as an immediate, otherwise the port number must be specified in DX.

IN AL, PORT (2 bytes)

(AL) \square (PORT)

IN AL, DX (1 byte)

(AL) \square (PDX)

IN AX, PORT (2 bytes)

(AL) \square (PORT), (AH) \square (PORT + 1)

IN AX, DX (1 byte)

(AL) \square (PDX), (AH) \square (PDX + 1)

OUT PORT, AL/AX—Output Data to Port

Flags affected: None.

Transfers byte in AL or word in AX to the specified hardware port address. If the port number is in the range of 0–255, it can be specified as an immediate. If the port number is greater than 255, it must be specified in DX. For example:

OUT PORT, AL (2 bytes)

(PORT) \square (AL)

OUT DX, AL (1 byte)

(PDX) \square (AL)

OUT PORT, AX (2 bytes)

(PORT) \square (AL), (PORT + 1) \square (AH)

OUT DX, AX (1 byte)

(PDX) \square (AL), (PDX + 1) \square (AH)

The instruction mentioned above can be easily understood with the help of the following examples.

EXAMPLE 6.1

Two memory locations R1 and R2 store 07H and 3FH respectively. Exchange the values in these memory locations without using the exchange instruction.

Solution:

DATA SEGMENT

R1 DB 07H

```

        R2      DB      3FH
DATA ENDS
CODE SEGMENT
ASSUME CS : CODE, DS: DATA
START:           MOV AX, DATA; Initialize the Data Segment
register
                MOV DS, AX
                MOV AL, R1; (AL) □ 07H
                MOV BL, R2; (BL) □ 3FH
                MOV R1, BL; (R1) □ (BL)
                MOV R2, AL; (RL) □ (AL)
CODE ENDS

```

If one uses the exchange instruction, the code segments will be

```

CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START:           MOV AX, DATA
                MOV DS, AX
                MOV AL, R1
                XCHG AL, R2
                MOV R1, AL
CODE ENDS

```

EXAMPLE 6.2

Two memory locations R1 and R2 store 07H and 3FH respectively. Exchange the values in these memory locations using indirect addressing.

Solution:

```

DATA SEGMENT
        R1      DB      07H
        R2      DB      3FH
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START:           MOV AX, DATA; Initialize the Data
Segment register
                MOV DS, AX
                LEA SI, R1; Offset address of R1 in SI
                LEA DI, R2; Offset address of R2 in DI
                MOV AL, (SI) ; (AL) □ (R1)
                MOV BL, (DI) ; (BL) □ (R2)

```

```

    MOV (SI), BL ; (R1) □ (BL)
    MOV (DI), AL ; (R2) □ (AL)
CODE ENDS

```

EXAMPLE 6.3

A string “MY NAME IS” is stored in memory as MES. The name in 14 characters is to be stored along with MES, to make it a 25 character string to look like:

MY NAME IS PRASHANT RISHI

Solution:

Use MOVSB instruction.

```

DATA SEGMENT
    MES      DB "MY NAME IS"; 11 character string
    NAME1    DB "PRASHANT RISHI"; 14 characters
    MES1     DB 25 DUP (0)
DATA ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA, ES: DATA
;
; Initialize Data Segment and Extra Segment registers
;
START:      MOV AX, DATA
            MOV DS, AX
            MOV ES, AX
            LEA SI, MES; Point SI to Source String
            LEA DI, MES1; Point DI to Destination String
            MOV CX, 11; move 11 to CX Counter Register
            CLD; Clear DF to auto increment SI and DI
REP      MOVSB
;
; Move NAME1 String to MES1. DI already points to address in
MES1 where NAME1
; must start
;
            LEA SI, NAME1
            MOV CX, 14
            CLD
REP      MOVSB
CODE ENDS

```

EXERCISES

1. What will be the values in K1, K2, K3 memory locations at the end of following program?

```
DATA SEGMENT
    K1      DW      3F99H
    K2      DW      7FFFH
    K3      DB      02H
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START:           MOV AX, DATA
                MOV DS, AX
                MOV AX, K1
                MOV BX, K2
                MOV CL, K3
                MOV AL, BL
                MOV BL, CL
                MOV CL, AL
                MOV K1, AX
                MOV K2, BX
                MOV K3, CL
CODE ENDS
```

2. Three 16-bit numbers N1, N2 and N3 are stored in memory. These numbers are to be changed in the following manner

- (a) Upper bytes of N1 and N2 are exchanged.
- (b) Lower bytes of N2 and N3 are exchanged.

Write separate programs using (i) register addressing, (ii) indirect addressing, and (iii) XCHG instructions.

3. Rewrite the program in Example 6.2 using LODSB and STOSB instructions.

6.8 ARITHMETIC GROUP

The 8086 instructions provide for

- the addition, subtraction, multiplication and division of binary numbers (signed and unsigned).
- the addition, subtraction, multiplication and division of ASCII or unpacked decimal numbers (one digit per byte).

- the addition and subtraction of packed BCD numbers (two digits per byte).
- increment/decrement by 1 for byte operands or for word operands stored in register or memory.
- the comparison of two bytes or words. One of the bytes/words may be an immediate or register value whereas the other byte/word must be either a register or a memory location.

In case the arithmetic operation on packed BCD or unpacked decimal numbers has been performed, the adjustment instructions (AAA, DAA, AAS, DAS, AAM and AAD) may be executed to put the result back in proper numeric format.

ADD dest, src—Arithmetic Addition

Flags affected: AF, CF, OF, PF, SF, ZF.

Adds “src” to “dest” and replaces the original contents of “dest”. Both the operands are binary.

Register to register operation

ADD RBD, RBS (2 bytes)

(RBD) \square (RBD) + (RBS)

ADD RWD, RWS (2 bytes)

(RWD) \square (RWD) + (RWS)

Register to memory operation

ADD DADDR, RB (2, 3 or 4 bytes)

(EA) \square (EA) + (RB)

ADD DADDR, RW (2, 3 or 4 bytes)

(EA) \square (EA) + (RW)

Memory to register operation

ADD RB, DADDR (2, 3 or 4 bytes)

(RB) \square (RB) + (EA)

ADD RW, DADDR (2, 3 or 4 bytes)

(RW) \square (RW) + (EA)

Immediate data to register operation

ADD RB, DATA8 (3 bytes)

(RB) \square (RB) + DATA8

ADD RW, DATA16 (3 or 4 bytes)

(RW) \square (RW) + DATA16

Immediate data to memory operation

ADD DADDR, DATA8 (3, 4 or 5 bytes)

(EA) \square (EA) + DATA8

ADD DADDR, DATA16 (3, 4, 5 or 6 bytes)

(EA) \square (EA) + DATA16

Immediate data to accumulator operation

ADD AL, DATA8 (2 bytes)

(AL) \square (AL) + DATA8

ADD AX, DATA16 (3 bytes)

(AX) \square (AX) + DATA16

ADC dest, src—Add with Carry

Flags affected: AF, CF, OF, SF, PF, ZF.

Sums two binary operands placing the result in the destination. If CF is set, a 1 is added to the destination.

Register to register operation

ADC RBD, RBS (2 bytes)

(RBD) \square (RBD) + (RBS) + (CF)

ADC RWD, RWS (2 bytes)

(RWD) \square (RWD) + (RWS) + (CF)

Register to memory operation

ADC DADDR, RB (2, 3 or 4 bytes)

(EA) \square (EA) + (RB) + (CF)

ADC DADDR, RW (2, 3 or 4 bytes)

(EA) \square (EA) + (RW) + (CF)

Memory to register operation

ADC RB, DADDR (2, 3 or 4 bytes)

(RB) \square (RB) + (EA) + (CF)

ADC RW, DADDR (2, 3 or 4 bytes)

(RW) \square (RW) + (EA) + (CF)

Immediate data to register operation

ADC RB, DATA8 (3 bytes)

(RB) \square (RB) + DATA8 + (CF)

ADC RW, DATA16 (3 or 4 bytes)

(RW) \square (RW) + DATA16 + (CF)

Immediate data to memory operation

ADC DADDR, DATA8 (3, 4 or 5 bytes)

(EA) \square (EA) + DATA8 + (CF)

ADC DADDR, DATA16 (3, 4, 5 or 6 bytes)

(EA) \square (EA) + DATA16 + (CF)

Immediate data to accumulator operation

ADC AL, DATA8 (2 byte)

(AL) \square (AL) + DATA8 + (CF)

ADC AX, DATA16 (3 bytes)

(AX) \square (AX) + DATA16 + (CF)

SUB dest, src—Subtract

Flags affected: AF, CF, OF, PF, SF, ZF.

The source is subtracted from the destination and the result is stored in the destination.

Register from register operation

SUB RBD, RBS (2 bytes)

(RBD) \square (RBD) – (RBS)

SUB RWD, RWS (2 bytes)

(RWD) \square (RWD) – (RWS)

Register from memory operation

SUB DADDR, RB (2, 3 or 4 bytes)

(EA) \square (EA) – (RB)

SUB DADDR, RW (2, 3 or 4 bytes)

(EA) \square (EA) – (RW)

Memory from register operation

SUB RB, DADDR (2, 3 or 4 bytes)

(RB) \square (RB) – (EA)

SUB RW, DADDR (2, 3 or 4 bytes)

(RW) \square (RW) – (EA)

Immediate data from register operation

SUB RB, DATA8 (3 bytes)

(RB) \square (RB) – DATA8

SUB RW, DATA16 (3 or 4 bytes)

(RW) \square (RW) – DATA16

Immediate data from memory operation

SUB DADDR, DATA8 (3, 4 or 5 bytes)

(EA) □ (EA) – DATA8

SUB DADDR, DATA16 (3, 4, 5 or 6 bytes)

(EA) □ (EA) – DATA16

Immediate data from accumulator operation

SUB AL, DATA8 (2 bytes)

(AL) □ (AL) – DATA8

SUB AX, DATA16 (3 bytes)

(AX) □ (AX) – DATA16

SBB dest, src—Subtract with Borrow/Carry

Flags affected: AF, CF, OF, PF, SF, ZF.

Subtracts the source from the destination, and subtracts 1 extra if the Carry Flag is set. The result is stored in “dest”.

Register from register operation

SBB RBD, RBS (2 bytes)

(RBD) □ (RBD) – (RBS) – (CF)

SBB RWD, RWS (2 bytes)

(RWD) □ (RWD) – (RWS) – (CF)

Register from memory operation

SBB DADDR, RB (2, 3 or 4 bytes)

(EA) □ (EA) – (RB) – (CF)

SBB DADDR, RW (2, 3 or 4 bytes)

(EA) □ (EA) – (RW) – (CF)

Memory from register operation

SBB RB, DADDR (2, 3 or 4 bytes)

(RB) □ (RB) – (EA) – (CF)

SBB RW, DADDR (2, 3 or 4 bytes)

(RW) □ (RW) – (EA) – (CF)

Immediate data from register operation

SBB RB, DATA8 (3 bytes)

(RB) □ (RB) – DATA8 – (CF)

SBB AX, DATA16 (3 bytes)

(AX) □ (AX) – DATA16 – (CF)

Immediate data from memory operation

SBB DADDR, DATA8 (3, 4 or 5 bytes)

(EA) □ (EA) – DATA8 – (CF)

SBB DADDR, DATA16 (3, 4, 5 or 6 bytes)

(EA) □ (EA) – DATA16 – (CF)

Immediate data from accumulator operation

SBB AL, DATA8 (2 bytes)

(AL) □ (AL) – DATA8 – (CF)

SBB RW, DATA16 (3 or 4 bytes)

(RW) □ (RW) – DATA16 – (CF)

MUL src—Unsigned Multiply

Flags affected: CF, OF (AF, PF, SF, ZF undefined).

Unsigned multiply of the accumulator by the source. Source may be a register or a memory location. If “src” is a byte value, then AL is used as the other multiplicand and the result is placed in AX. If “src” is a word value, then AX is multiplied by “src” and DX:AX receives the result. For example:

MUL RB (2 bytes)

(AX) □ (AL) □ (RB)

MUL RW (2 bytes)

(DX)(AX) □ (AX) □ (RW)

MUL DADDR (2, 3 or 4 bytes)

(AX) □ (AL) □ (EA)

MUL DADDR (2, 3 or 4 bytes)

(DX)(AX) □ (AX) □ (EA)

IMUL src—Signed Multiply

Flags affected: CF, OF (AF, PF, SF, ZF undefined).

Signed multiplication of accumulator by “src” with the result placed in the accumulator. Source may be a register or a memory location. If the source operand is a byte value, it is multiplied by AL and the result is stored in AX. If the source operand is a word value it is multiplied by AX and the result is stored in DX:AX. For example:

IMUL RB (2 bytes)

(AX) □ (AL) □ (RB)

IMUL RW (2 bytes)

(DX)(AX) □ (AX) □ (RW)

IMUL DADDR (2, 3 or 4 bytes)

(AX) □ (AL) □ (EA)

IMUL DADDR (2, 3 or 4 bytes)

(DX)(AX) □ (AX) □ (EA)

DIV src—Divide

Flags affected: (AF, CF, OF, PF, SF, ZF undefined).

Unsigned binary division of the accumulator by source. Source may be a register or a memory location. If the source divisor is a byte value, AX is divided by “src” and the quotient is placed in AL and the remainder in AH. If the source operand is a word value, DX:AX is divided by “src” and the quotient is stored in AX and the remainder in DX. For example:

DIV RB (2 bytes)

(AX) □ (AX)/(RB)

DIV RW (2 bytes)

(DX)(AX) □ (DX)(AX)/(RW)

DIV DADDR (2, 3 or 4 bytes)

(AX) □ (AX)/(EA)

DIV DADDR (2, 3 or 4 bytes)

(DX)(AX) □ (DX)(AX)/(EA)

IDIV src—Signed Integer Division

Flags affected: (AF, CF, OF, PF, SF, ZF undefined).

Signed binary division of the accumulator by source. Source may be a register or a memory location. If the source is a byte value, AX is divided by “src” and the quotient is stored in AL and the remainder in AH. If the source is a word value, DX:AX is divided by “src”, and the quotient is stored in AL and the remainder in DX. For example:

IDIV RB (2 bytes)

(AX) □ (AX)/(RB)

IDIV RW (2 bytes)

(DX)(AX) □ (DX)(AX)/(RW)

IDIV DADDR (2, 3 or 4 bytes)

(AX) □ (AX)/(EA)

IDIV DADDR (2, 3 or 4 bytes)

(DX)(AX) □ (DX)(AX)/(EA)

INC dest—Increment

Flags affected: AF, OF, PF, SF, ZF.

Adds 1 to the destination unsigned binary operand. The destination may be an 8-bit value in memory/register or it may be a 16-bit value in memory/register. For example:

INC DADDR (2, 3 or 4 bytes)

$(EA) \square (EA) + 1$

INC RB (2 bytes)

$(RB) \square (RB) + 1$

INC RW (2 bytes)

$(RW) \square (RW) + 1$

DEC dest—Decrement

Flags affected: AF, OF, PF, SF, ZF.

Unsigned binary subtraction of 1 from the destination. The destination may be an 8-bit value in memory/register or it may be a 16-bit value in memory/register. For example:

DEC DADDR (2, 3 or 4 bytes)

$(EA) \square (EA) - 1$

DEC RB (2 bytes)

$(RB) \square (RB) - 1$

DEC RW (2 bytes)

$(RW) \square (RW) - 1$

CMP dest, src—Compare

Flags affected: AF, CF, OF, PF, SF, ZF.

Subtracts the source from the destination and updates the flags, but does not save the result. Flags can subsequently be checked for conditions.

Register to register operation

CMP RBD, RBS (2 bytes)

$(RBD) - (RBS)$

CMP RWD, RWS (2 bytes)

$(RWD) - (RWS)$

Register to memory operation

CMP DADDR, RB (2, 3 or 4 bytes)

$(EA) - (RB)$

CMP DADDR, RW (2, 3 or 4 bytes)

$(EA) - (RW)$

Memory to register operation

CMP RB, DADDR (2, 3 or 4 bytes)

(RB) – (EA)

CMP RW, DADDR (2, 3 or 4 bytes)
(RW) - (EA)

Immediate data to register operation

CMP RB, DATA8 (3 bytes)
(RB) – DATA8

CMP RW, DATA16 (3 or 4 bytes)
(RW) – DATA16

Immediate data to memory operation

CMP DADDR, DATA8 (3, 4 or 5 bytes)
(EA) – DATA8

CMP DADDR, DATA16 (3, 4, 5 or 6 bytes)
(EA) – DATA16

Immediate data to accumulator operation

CMP AL, DATA8 (2 bytes)
(AL) – DATA8

CMP AX, DATA16 (3 bytes)
(AX) – DATA16

CMPSB/CMPSW dest, src—Compare String (Byte or Word)

Flags affected: AF, CF, OF, PF, SF, ZF.

Subtracts the destination string value from the source without saving the results. Updates the flags based on the subtraction and the index registers SI and DI are incremented or decremented depending on the state of the Direction Flag. CMPSB increments/decrements the index registers by 1 and CMPSW increments/decrements the index registers by 2. The REP prefixes can be used to process the entire data items. For source string using SI, data segment is used whereas for destination string using DI, extra segment is used. For example:

CMPSB (1 byte)

$$(\text{DS:SI}) - (\text{ES:DI}), (\text{DI}) \square (\text{DI}) + 1, (\text{SI}) \square (\text{SI}) + 1 \text{ if } (\text{DF}) = 0, \\ (\text{DI}) \square (\text{DI}) - 1, (\text{SI}) \square (\text{SI}) - 1 \text{ if } (\text{DF}) = 1$$

CMPSW (1 byte)

$$(DS:SI) - (ES:DI), (DI) \square (DI) + 2, (SI) \square (SI) + 2 \text{ if } (DF) = 0, \\ (DI) \square (DI) - 2, (SI) \square (SI) - 2 \text{ if } (DF) = 1$$

SCASB/SCASW—Scan String (Byte or Word)

Flags affected: AF, CF, OF, PF, SF, ZF.

Compares the value at ES:DI from the accumulator and sets the flags similar to a subtraction. DI is incremented/decremented based on the instruction format (or operand size) and the state of the Direction Flag. Use with REP prefixes. For example:

SCASB (1 byte)

$$(AL) - (ES:DI), \quad (DI) \square (DI) + 1 \text{ if } (DF) = 0, \\ (DI) \square (DI) - 1 \text{ if } (DF) = 1$$

SCASW (1 byte)

$$(AX) - (ES:DI), \quad (DI) \square (DI) + 2 \text{ if } (DF) = 0, \\ (DI) \square (DI) - 2 \text{ if } (DF) = 1$$

CBW—Convert Byte to Word

Flags affected: None.

Converts byte in AL to word value in AX by extending the sign of AL throughout the register AH.

CWD—Convert Word to Double word

Flags affected: None.

Extends the sign of word in the register AX throughout the register DX forming a double word quantity in DX:AX.

DAA (1 byte)—Decimal Adjust for Addition

Flags affected: AF, CF, PF, SF, ZF, (OF undefined).

Corrects the result (in AL) of a previous BCD addition operation. Contents of AL are changed to a pair of packed decimal digits.

DAS (1 byte)—Decimal Adjust for Subtraction

Flags affected: AF, CF, PF, SF, ZF, (OF undefined).

Corrects the result (in AL) of a previous BCD subtraction operation. Contents of AL are changed to a pair of packed decimal digits.

AAA (1 byte)—ASCII Adjust for Addition

Flags affected: AF, CF, (OF, PF, SF, ZF undefined).

Changes the contents of AL to a valid unpacked decimal. The high-order nibble is zeroed.

AAD (2 bytes)—ASCII Adjust for Division

Flags affected: SF, ZF, PF (AF, CF, OF undefined).

Used before dividing the unpacked decimal numbers. Multiplies AH by 10 and then adds the result into AL. Sets AH to zero. For example:

$$(AL) \square (AH) \square 10 + (AL)$$

$$(AH) \square 0$$

AAM (2 bytes)—ASCII Adjust for Multiplication
Flags affected: PF, SF, ZF, (AF, CF, OF undefined).

(AH) := (AL)/10

(AL) := (AL) mod 10

Used after the multiplication of two unpacked decimal numbers, this instruction adjusts an unpacked decimal number. The high-order nibble of each byte must be zeroed before using this instruction.

AAS (1 byte)—ASCII Adjust for Subtraction

Flags affected: AF, CF, (OF, PF, SF, ZF undefined).

Corrects the result of a previous unpacked decimal subtraction in AL. The high-order nibble is zeroed.

Following are some examples to understand these instructions better.

EXAMPLE 6.4

Multiply two 8-bit numbers stored in P1 and P2 and store the result in P3, using both direct and indirect addressing.

Solution:

(a) Using Direct addressing

DATA SEGMENT

P1	DB	7FH
P2	DB	05H
P3	DW	0000H

DATA ENDS

CODE SEGMENT

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

MOV AL, P1

MUL BYTE PTR P2; Result is in AX

MOV P3, AX

CODE ENDS

(b) Using Indirect addressing

In case we use indirect addressing, the code segment will change to

CODE SEGMENT

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

LEA SI, P1

```

LEA DI, P2
MOV AL, (SI)
MUL BYTE PTR (DI)
LEA DI, P3
MOV (DI), AX
CODE ENDS

```

EXAMPLE 6.5

Two 8-bit numbers X and Y are stored in memory. Calculate Z based on the following equation and store it in memory as a 16-bit number. Discard the remainder in division operation.

$$Z = (X^2 + Y^2) / (X^2 - Y^2)$$

Solution:

```

DATA SEGMENT
    X        DB      05H
    Y        DB      07H
    Z        DW      0000H
    SUM-SQ   DW      0000H
    DIF-SQ   DW      0000H
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START:    MOV AX, DATA
          MOV DS, AX
          MOV AL, X
          MUL X
          MOV BX, AX; (BX) □ X2
          MOV AL, Y
          MUL Y
          MOV CX, AX; (CX) □ Y2
          MOV AX, BX; (AX) □ X2
          ADD AX, CX; (AX) □ X2 + Y2
          MOV SUM-SQ, AX
          MOV AX, BX
          SUB AX, CX; (AX) □ X2 - Y2
          MOV DIF-SQ, AX
          MOV DX, 0000H

```

```

MOV AX, SUM-SQ; (AX)  $\square$  X2 + Y2
DIV DIF-SQ
MOV Z, AX
CODE ENDS

```

EXERCISES

1. What will be the value in S3 and Flag Register at the end of the following program?

```

DATA SEGMENT
    S1      DW      0735H
    S2      DW      0826H
    S3      DB      05H
DATA ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
START:   MOV AX, DATA
        MOV DS, AX
        MOV AX, S1
        ADD AX, S2
        MOV S1, AX
        SUB AX, 00FFH
        DIV S3
        MOV S3, AL
CODE ENDS

```

2. Two memory locations A1 and A2 have packed BCD numbers 75 and 29 respectively. Write a program to add these, convert the result to packed BCD format and store it in the memory location A3.
3. Rewrite the program in Example 6.5, to consider the remainder of division operation. If the remainder is equal to or more than 0.5, then round off the value of Z by adding 1.

6.9 LOGICAL GROUP

The logical operations such as AND, OR, NOT, XOR may be performed on 8-bit or 16-bit operands. The ‘shift’ and ‘rotate’ instructions provide for right or left shift or rotate by number of bits specified in the CL register. ‘Test’ is another interesting instruction using which two 16-bit word operands may be ANDed, and based on the result the status flags

are modified. The operands, however, are not altered by these instructions.

AND dest, src—Logical AND

Flags affected: CF, OF, PF, SF, ZF (AF undefined).

Performs a logical AND of the two operands replacing the destination with the result.

Register to register operation

AND RBD, RBS (2 bytes)

(RBD) \square (RBD) AND (RBS)

AND RWD, RWS (2 bytes)

(RWD) \square (RWD) AND (RWS)

Register to memory operation

AND DADDR, RB (2, 3 or 4 bytes)

(EA) \square (EA) AND (RB)

AND DADDR, RW (2, 3 or 4 bytes)

(EA) \square (EA) AND (RW)

Memory to register operation

AND RB, DADDR (2, 3 or 4 bytes)

(RB) \square (RB) AND (EA)

AND RW, DADDR (2, 3 or 4 bytes)

(RW) \square (RW) AND (EA)

Immediate data to register operation

AND RB, DATA8 (3 bytes)

(RB) \square (RB) AND DATA8

AND RW, DATA16 (4 bytes)

(RW) \square (RW) AND DATA16

Immediate data to memory operation

AND DADDR, DATA8 (3, 4 or 5 bytes)

(EA) \square (EA) AND DATA8

AND DADDR, DATA16 (4, 5 or 6 bytes)

(EA) \square (EA) AND DATA16

Immediate data to accumulator operation

AND AL, DATA8 (2 bytes)

(AL) \square (AL) AND DATA8

AND AX, DATA16 (3 bytes)
(AX) \square (AX) AND DATA16
OR dest, src—Inclusive Logical OR
Flags affected: CF, OF, PF, SF, ZF, (AF undefined).
Logical inclusive OR of the two operands returning the result in the destination. Any bit set in either operand will be set in the destination.

Register to register operation

OR RBD, RBS (2 bytes)
(RBD) \square (RBD) OR (RBS)
OR RWD, RWS (2 bytes)
(RWD) \square (RWD) OR (RWS)

Register to memory operation

OR DADDR, RB (2, 3 or 4 bytes)
(EA) \square (EA) OR (RB)
OR DADDR, RW (2, 3 or 4 bytes)
(EA) \square (EA) OR (RW)

Memory to register operation

OR RB, DADDR (2, 3 or 4 bytes)
(RB) \square (RB) OR (EA)
OR RW, DADDR (2, 3 or 4 bytes)
(RW) \square (RW) OR (EA)

Immediate data to register operation

OR RB, DATA8 (3 bytes)
(RB) \square (RB) OR DATA8
OR RW, DATA16 (4 bytes)
(RW) \square (RW) OR DATA16

Immediate data to memory operation

OR DADDR, DATA8 (3, 4 or 5 bytes)
(EA) \square (EA) OR DATA8
OR DADDR, DATA16 (4, 5 or 6 bytes)
(EA) \square (EA) OR DATA16

Immediate data to accumulator operation

OR AL, DATA8 (2 bytes)
(AL) \square (AL) OR DATA8
OR AX, DATA16 (3 bytes)

$(AX) \square (AX) \text{ OR } \text{DATA16}$
XOR dest, src—Exclusive OR
Flags affected: CF, OF, PF, SF, ZF (AF undefined).
Performs a bit-wise exclusive OR of the operands and returns the result to the destination.

Register to register operation

XOR RBD, RBS (2 bytes)
 $(RBD) \square (RBD) \text{ XOR } (RBS)$
XOR RWD, RWS (2 bytes)
 $(RWD) \square (RWD) \text{ XOR } (RWS)$

Register to memory operation

XOR DADDR, RB (2, 3 or 4 bytes)
 $(EA) \square (EA) \text{ XOR } (RB)$
XOR DADDR, RW (2, 3 or 4 bytes)
 $(EA) \square (EA) \text{ XOR } (RW)$

Memory to register operation

XOR RB, DADDR (2, 3 or 4 bytes)
 $(RB) \square (RB) \text{ XOR } (EA)$
XOR RW, DADDR (2, 3 or 4 bytes)
 $(RW) \square (RW) \text{ XOR } (EA)$

Immediate data to register operation

XOR RB, DATA8 (3 bytes)
 $(RB) \square (RB) \text{ XOR } \text{DATA8}$
XOR RW, DATA16 (4 bytes)
 $(RW) \square (RW) \text{ XOR } \text{DATA16}$

Immediate data to memory operation

XOR DADDR, DATA8 (3, 4 or 5 bytes)
 $(EA) \square (EA) \text{ XOR } \text{DATA8}$
XOR DADDR, DATA16 (4, 5 or 6 bytes)
 $(EA) \square (EA) \text{ XOR } \text{DATA16}$

Immediate data to accumulator operation

XOR AL, DATA8 (2 bytes)
 $(AL) \square (AL) \text{ XOR } \text{DATA8}$
XOR AX, DATA16 (3 bytes)
 $(AX) \square (AX) \text{ XOR } \text{DATA16}$

TEST dest, src—Test for Bit Pattern

Flags affected: CF, OF, PF, SF, ZF (AF undefined).

Performs a logical AND of the two operands updating the flag register without saving the result.

Register and register operation

TEST RBD, RBS (2 bytes)

(RBD) AND (RBS)

TEST RWD, RWS (2 bytes)

(RWD) AND (RWS)

Register and memory operation

TEST DADDR, RB (2, 3 or 4 bytes)

(EA) AND (RB)

TEST DADDR, RW (2, 3 or 4 bytes)

(EA) AND (RW)

Register and immediate data operation

TEST RB, DATA8 (3 bytes)

(RB) AND DATA8

TEST RW, DATA16 (4 bytes)

(RW) AND DATA16

Memory and immediate data operation

TEST DADDR, DATA8 (3, 4 or 5 bytes)

(EA) AND DATA8

TEST DADDR, DATA16 (4, 5 or 6 bytes)

(EA) AND DATA16

Accumulator and immediate data operation

TEST AL, DATA8 (2 bytes)

(AL) AND DATA8

TEST AX, DATA16 (3 bytes)

(AX) AND DATA16

NOT dest—1’s Compliment Negation (Logical NOT)

Flags affected: None.

Inverts the bits of the “dest” operand forming the 1’s complement. The destination may be a register or a memory location. For example:

NOT DADDR (2, 3 or 4 bytes)

(EA) □ NOT (EA)

NOT RB (2 bytes)

(RB) \square NOT (RB)

NOT RW (2 bytes)

(RW) \square NOT (RW)

NEG dest—2's Complement Negation

Flags affected: AF, CF, OF, PF, SF, ZF.

Subtracts the destination from 0 and saves the 2's complement of “dest” back into “dest”. The destination may be a register or a memory location.

For example:

NEG DADDR (2, 3 or 4 bytes)

(EA) \square NOT (EA) + 1

NEG RB (2 bytes)

(RB) \square NOT (RB) + 1

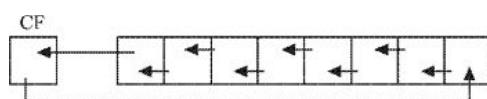
NEG RW (2 bytes)

(RW) \square NOT (RW) + 1

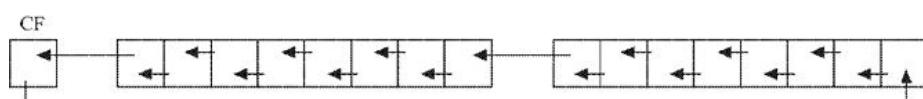
RCL dest, count—Rotate through Carry Left

Flags affected: CF, OF.

Rotates the bits in the destination to the left “count” times through the Carry Flag with all the data pushed out from the left side and re-entering on the right. The Carry Flag holds the last bit rotated out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register. For example:



or



RCL DADDR, N (2, 3 or 4 bytes)

Rotates the contents of the data memory location addressed by DADDR left through the Carry Flag. If N = 1, then rotates 1 bit position. If N = CL, then the register CL contents provide the number of bit positions. DADDR may address a byte or a word.

RCL RB/RW, N (2 bytes)

Rotates left through the Carry Flag, the 8-bit contents of the RB register, or the 16-bit contents of RW register, as illustrated for memory operate.

RCR dest, count—Rotate through Carry Right

Flags affected: CF, OF.

Rotates the bits in the destination to the right “count” times through the Carry Flag with all the data pushed out from the right side, re-entering on the left. The Carry Flag holds the last bit rotated out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register. For example:

RCR DADDR, N (2, 3 or 4 bytes)

Rotates the contents of the data memory location addressed by DADDR right through the Carry Flag. If N = 1, then rotates 1 bit position. If N = CL, then register CL contents provide the number of bit positions. DADDR may address a byte or a word.

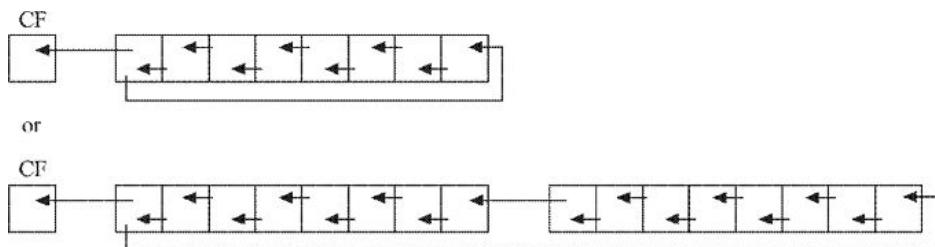
RCR RB/RW, N (2 bytes)

Rotates right through the Carry Flag, the 8-bit contents of RB register, or the 16-bit contents of RW register, as illustrated for memory operate.

ROL dest, count—Rotate Left

Flags affected: CF, OF.

Rotates the bits in the destination to the left “count” times with all the data pushed out from the left side, re-entering on the right. The Carry Flag will contain the value of the last bit rotated out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register. For example:



ROL DADDR, N (2, 3 or 4 bytes)

Rotates the contents of the data memory location addressed by DADDR left. Moves the leftmost bit into the Carry Flag. If N = 1, then rotates 1 bit position. If N = CL, then the register CL contents provide the number of bit positions. DADDR may address a byte or a word.

ROL RB/RW, N (2 bytes)

Rotates left the 8-bit contents of the RB register, or the 16-bit contents of the RW register, as illustrated for memory operate.

ROR dest, count—Rotate Right

Flags affected: CF, OF.

Rotates the bits in the destination to the right “count” times with all the data pushed out from the right side, re-entering on the left. The Carry Flag will contain the value of the last bit rotated out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register.

ROR DADDR, N (2, 3 or 4 bytes)

Rotates the contents of the data memory location addressed by DADDR right. Moves the rightmost bit into the Carry Flag. If N = 1, then rotates 1 bit position. If N = CL, then the register CL contents provide the number of bit positions. DADDR may address a byte or a word.

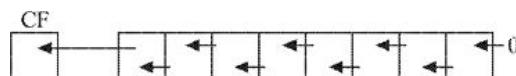
ROR RB/RW, N (2 bytes)

Rotates right the 8-bit contents of the RB register, or the 16-bit contents of the RW register, as illustrated for memory operate.

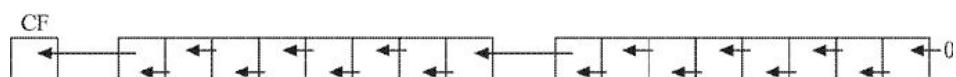
SAL/SHL dest, count—Shift Arithmetic Left/Shift Logical Left

Flags affected: CF, OF, PF, SF, ZF (AF undefined).

Shifts left the destination by “count” bits with zeroes shifted to right. The Carry Flag contains the last bit shifted out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register. For example:



or



SAL/SHL DADDR, N (2, 3 or 4 bytes)

Shifts left the contents of the data memory location addressed by DADDR. Moves the leftmost bit into the Carry Flag. If N = 1, then shifts 1 bit position. If N = CL, then the register CL contents provide the number of bit positions. DADDR may address a byte or word.

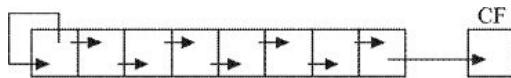
SAL/SHL RB/RW, N (2 bytes)

Shift left the 8-bit contents of the RB register, or the 16-bit contents of the RW register, as illustrated for memory operate.

SAR dest, count—Shift Arithmetic Right

Flags affected: CF, OF, PF, SF, ZF (AF undefined).

Shifts the destination right by “count” bits with the current sign bit replicated in the leftmost bit. The Carry Flag contains the last bit shifted out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register. For example:



or



SAR DADDR, N (2, 3 or 4 bytes)

Shifts right the contents of the data memory location by “count” bits with the current sign bit replicated in the leftmost bit. The Carry Flag contains the last bit shifted out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register.

SAR RB/RW, N (2 bytes)

Shifts right the 8-bit contents of RB register, or the 16-bit contents of RW register, as illustrated for memory operate.

SHR dest, count—Shift Logical Right

Flags affected: CF, OF, PF, SF, ZF (AF undefined).

Shifts right the destination by “count” bits with zeroes shifted to the left. The Carry Flag contains the last bit shifted out. The destination may be a register or a memory location. The count may be 1 or the contents of the CL register. For example:

SHR DADDR, N (2, 3 or 4 bytes)

Shifts right the contents of the data memory location addressed by DADDR. Moves the rightmost bit into the Carry Flag. If N = 1, then shifts 1 bit position. If N = CL, then the register CL contents provide the number of bit positions. DADDR may address a byte, or word.

SHR RB/RW, N (2 bytes)

Shifts right the 8-bit contents of the RB register, or the 16-bit contents of the RW register, as illustrated for memory operate.

The following examples will help in understanding the logical group instructions.

EXAMPLE 6.6

Two numbers X1 and X2 are stored in memory. Perform the following operations and store the results in X3.

Solution:

```
(X3) = ((X1) AND 0FH) XOR (X2)

DATA SEGMENT
    X1    DB    9AH
    X2    DB    F7H
    X3    DB    00H
DATA ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV AL, X1
            LEA SI, X2
            AND AL, 0FH
            XOR AL, (SI)
            MOV X3, AL
CODE ENDS
```

EXAMPLE 6.7

The numbers in ASCII codes are stored in A1 and A2. Subtract A2 from A1, convert the result in ASCII and store in A3.

Solution:

```
DATA SEGMENT
    A1    DB    39H: ASCII CODE for 9
    A2    DB    34H: ASCII CODE for 4
    A3    DB    00H
DATA ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV AL, A1
            SUB AL, A2
            AAS; ASCII adjust for subtraction
            OR 30H; To convert back into ASCII
            MOV A3, AL
CODE ENDS
```

EXERCISES

1. What will be the values of X1 and X2 at the end of following program?

```
DATA SEGMENT
    X1    DB    07H
    X2    DB    F0H
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START:    MOV    AX, DATA
          MOV    DS, AX
          MOV    AL, X1
          XOR    AL, 9FH
          MOV    CL, 02
          ROR    AL, CL
          MOV    X1, AL
          AND    AL, X2
          MOV    CL, 03
          SHL    AL, CL
          MOV    X2, AL
CODE ENDS
```

2. Two numbers N1 and N2 are stored in memory. Develop a program to perform the following operations and store the results back in N1 and N2.

$$N1 = (N1 \square 64) \text{ AND } 0FH$$

$$N2 = (N2/16) \text{ XOR } 0FH$$

Do not use multiplication or division instruction.

3. An 8-bit number is stored in memory. Write a program to change the bit pattern as follows:

7th bit comes to 2nd bit

6th bit comes to 1st bit

4th bit comes to 5th bit

6.10 CONTROL TRANSFER GROUP

Standard operations like Unconditional jump, Conditional jump, Call and Return are supported in the 8086 instructions. Unlike the 8085, the 8086 does not provide for Conditional call and Conditional return operations. The 8086 provides for a complex control through loop instructions.

These instructions (LOOP, LOOPZ, LOOPE, LOOPNE, LOOPNZ) cause the contents of the CX register to be decremented by 1, and

- in case of LOOP instruction if (CX) \neq 0 the 8086 jumps to an instruction in the range between +127 and -128 bytes. It is equivalent to

DEC CX

JNZ label instruction sequence.

- in case of LOOPZ or LOOPE instruction, the 8086 performs jump if (CX) \neq 0 and Zero Flag is set (due to the comparison of two operands).
- in case of LOOPNE or LOOPNZ instruction, the 8086 performs jump if (CX) \neq 0 and Zero Flag is clear (i.e., two operands being compared are not equal).

It must be noted that the decrement of CX does not cause the modification of Zero Flag in LOOP instructions.

JMP LABEL—Unconditional Jump

Flags affected: None.

Unconditionally transfers control to “LABEL.” Jumps by default are within -32768 to 32767 bytes from the instructions following the jumps. NEAR and SHORT jumps cause the IP to be updated while the FAR jumps cause CS and IP to be updated. For example:

JMP LABEL (2 or 3 bytes)—Direct Addressing
(IP) \square (IP) + DISP

Jumps direct to program memory locations identified by LABEL. The displacement DISP which must be added to the Instruction Pointer will be computed as an 8-bit or a 16-bit signed binary number, as needed, by the assembler.

JMP LABEL (5 bytes)
(IP) \square DATA16, (CS) \square DATA16

Jumps direct into a new segment. It is FAR jump. The LABEL must be declared FAR through a directive. The assembler will calculate the code segment base address and the offset of LABEL from the segment base. These values will be placed as part of the instruction. These are loaded during the execution in CS and IP registers.

JMP DADDR (2, 3 or 4 bytes)—Indirect Addressing
(IP) \square (EA)

Jumps indirect in the current segment. The 16-bit contents of the data memory word addressed by DADDR are loaded into IP.

JMP DADDR, CS (2, 3 or 4 bytes)

(IP) \square (EA), (CS) \square (EA + 2)

Jumps indirect into a new segment. The 16-bit contents of the data memory word addressed by DADDR are loaded into IP. The contents of the next sequential 16-bit data memory word are loaded into the CS segment register.

JMP RW (2 bytes)

(IP) \square (RW)

Jumps to memory location whose address is contained in register RW.

J < COND. > LABEL—Conditional Jump. (Jump to LABEL if <COND.> is true.)

Flags affected: None.

The condition <COND.> is determined by status flags. Usually, the conditional jump occurs after an arithmetic/logic or a compare instruction. This will set the status flags. These instructions are limited to 8-bit signed displacement. Thus LABEL should occur within 256 bytes (-128 to +127 bytes) of the instruction. In all cases:

If <COND.> = True, then (IP) \square (IP) + DISP8
else continue

The following table describes the various mnemonics, their meanings and the jump conditions in case of conditional jump instructions.

Mnemonic	Meaning	Jump condition
JA	Jump if Above	(CF) = 0 and (ZF) = 0
JAE	Jump if Above or Equal	(CF) = 0
JB	Jump if Below	(CF) = 1
JBE	Jump if Below or Equal	(CF) = 1 or (ZF) = 1
JC	Jump if Carry	(CF) = 1
JCXZ	Jump if CX Zero	(CX) = 0
JE	Jump if Equal	(ZF) = 1
JG	Jump if Greater (signed)	(ZF) = 0 and (CF) = (OF)
JGE	Jump if Greater or Equal (signed)	(SF) = (OF)
JL	Jump if Less (signed)	(SF) \square (OF)
JLE	Jump if Less or Equal (signed)	(ZF) = 1 or (SF) \square (OF)
JNA	Jump if Not Above	(CF) = 1 or (ZF) = 1
JNAE	Jump if Not Above or Equal	(CF) = 1
JNB	Jump if Not Below	(CF) = 0
JNBE	Jump if Not Below or Equal	(CF) = 0 and (ZF) = 0

JNC	Jump if Not Carry	(CF) = 0
JNE	Jump if Not Equal	(ZF) = 0
JNG	Jump if Not Greater (signed)	(ZF) = 1 or (SF) \square (OF)
JNGE	Jump if Not Greater or Equal (signed)	(SF) \square (OF)
JNL	Jump if Not Less (signed)	(SF) = (OF)
JNLE	Jump if Not Less or Equal (signed)	(ZF) = 0 and (CF) = (OF)
JNO	Jump if Not Overflow (signed)	(OF) = 0
JNP	Jump if No Parity	(PF) = 0
JNS	Jump if Not Signed (signed)	(SF) = 0
JNZ	Jump if Not Zero	(ZF) = 0
JO	Jump if Overflow (signed)	(OF) = 1
JP	Jump if Parity	(PF) = 1
JPE	Jump if Parity Even	(PF) = 1
JPO	Jump if Parity Odd	(PF) = 0
JS	Jump if Signed (signed)	(SF) = 1
JZ	Jump if Zero	(ZF) = 1

As an example:

JA LABEL Jump if Above (2 bytes)

When (CF) = 0 and (ZF) = 0

(IP) \square (IP) + DISP8

else continue

It may be noted that Above, Equal and Below are used with respect to comparison or arithmetic operations using unsigned numbers whereas Greater, Equal and Less are used with comparison or arithmetic operations, using signed numbers.

It may also be noted that the number of conditions for branch are represented by two or more mnemonics. For example, the following mnemonics perform the same operations.

- JNA and JBE
- JNAE, JC and JB
- JNB, JNC and JAE
- JNBE and JA
- JNG and JLE
- JNGE and JL
- JNL and JGE
- JNLE and JG
- JNZ and JNE

JPE and JP
JPO and JNP
JZ and JE

It is a good programming practice to organize codes so that the expected case is executed without a jump since the actual jump takes longer to execute than failing through the test.

JCXZ LABEL—Jump if Register CX is zero

Flags affected: None.

Causes execution to branch to LABEL if register CX is zero. Uses unsigned comparison.

LOOP LABEL (2 bytes)—Decrement CX and Loop if CX is not zero

Flags affected: None.

Decrements CX by 1 and transfers control to LABEL if CX is not zero. The LABEL operand must be within -128 or 127 bytes of the instruction following the loop instruction.

(CX) \square (CX) – 1. If (CX) \square 0, then (IP) \square (IP) + DISP8

LOOPE/LOOPZ LABEL (2 bytes)—Loop While Equal/Loop While Zero

Flags affected: None.

Decrements CX by 1 (without modifying the flags) and transfers control to LABEL if (CX) \square 0 and the Zero Flag is set. The LABEL operand must be within -128 or 127 bytes of the instruction following the loop instruction.

(CX) \square (CX) – 1. If (CX) \square 0 and (ZF) = 1, then (IP) \square (IP) + DISP8

LOOPNZ/LOOPNE LABEL (2 bytes)—Loop While Not Zero/Loop While Not Equal

Flags affected: None.

Decrements CX by 1 (without modifying the flags) and transfers control to “LABEL” if (CX) \square 0 and the Zero Flag is clear. The “LABEL” operand must be within -128 or 127 bytes of the instruction following the loop instruction.

(CX) \square (CX) – 1. If (CX) \square 0 and (ZF) = 0, then (IP) \square (IP) + DISP8

CALL PROC-NAME—Procedure Call

Flags affected: None

Pushes the Instruction Pointer (and Code Segment for FAR calls) onto stack and loads the Instruction Pointer with the address of label PROC-NAME. Code continues with the execution at CS:IP. For example:

CALL PROC-NAME (3 bytes)

((SP)) □ (IP), (SP) □ (SP) – 2, (IP) □ (IP) + DISP

Calls a subroutine in the current program segment using direct addressing. Since the procedure PROC-NAME is resident in the same segment, it is NEAR call.

CALL PROC-NAME (5 bytes)

((SP)) □ (CS), (SP) □ (SP) – 2, ((SP)) □ (IP), (SP) □ (SP) – 2, (IP) □ DATA16, (CS) □ DATA16

Calls a procedure in another program segment (FAR call) using direct addressing. The PROC-NAME procedure must be declared FAR at the start using the directive “PROC-NAME PROC FAR”. The assembler will determine the code segment base for the segment which contains the procedure and the offset of the start of the procedure from the segment base. The assembler will place these values as part of the instruction code.

CALL DADDR (2, 3 or 4 bytes)

((SP)) □ (IP), (SP) □ (SP) – 2, (IP) □ (EA)

Calls a subroutine in the current program segment using indirect addressing. The address of the subroutine called is stored in the 16-bit data memory word addressed by DADDR.

CALL DADDR, CS (2, 3 or 4 bytes)

((SP)) □ (CS), (SP) □ (SP) – 2, ((SP)) □ (IP), (SP) □ (SP) – 2, (IP) □ (EA), (CS) □ (EA + 2)

Calls a subroutine in a different program segment using indirect addressing. The address of the subroutine called is stored in the 16-bit data memory word addressed by DADDR. The new CS register contents are stored in the next sequential program memory word.

CALL RW (2 bytes)

((SP)) □ (IP), (SP) □ (SP) – 2, (IP) □ (RW)

Calls a subroutine whose address is contained in the register RW.

RET nBytes—Return from Procedure

Flags affected: None.

Transfers control from a procedure back to the instruction address saved on the stack. “n bytes” is an optional number of bytes to be removed from the stack. FAR returns pop the IP followed by the CS, while the NEAR returns pop only the IP register. For example:

RET (1 byte)

(IP) □ ((SP)), (SP) □ (SP) + 2 for NEAR calls

Returns from a subroutine in the current segment.
 or
 $(IP) \square ((SP)), (SP) \square (SP) + 2, (CS) \square ((SP)), (SP) \square (SP) + 2$ for FAR calls
 Returns from a subroutine in another segment.

RET DATA16 (3 bytes)
 $(IP) \square ((SP)), (SP) \square (SP) + 2 + DATA16$
 Returns from a subroutine in the current segment and adds an immediate displacement to SP.

RET CS DATA16
 $(IP) \square ((SP)), (SP) \square (SP) + 2, (CS) \square ((SP)), (SP) \square (SP) + 2 + DATA16$
 Returns from a subroutine in another segment and adds an immediate displacement to SP.

Some examples are given below for better understanding of the control transfer group of instructions.

EXAMPLE 6.8

Develop a program to transfer 50 bytes of data from memory location starting from 2000H to 3000H, using the string instruction MOVSB.

Solution:

```

DATA SEGMENT
    X1    EQU    2000H
    X2    EQU    3000H
    CNT   EQU    50
DATA ENDS

CODE SEGMENT
    ASSUME CS: CODE, DS: DATA, ES: EXTRA
START:    MOV AX, DATA
          MOV DS, AX
          MOV ES, AX
          LEA SI, X1
          LEA DI, X2
          CLD
          MOV CX, CNT
          REP    MOVSB
CODE ENDS
  
```

EXAMPLE 6.9

Develop a program to compare two arrays of 10 elements each. If they are same, store 1 at memory location INDT otherwise store 0.

Solution:

```
DATA SEGMENT
    ARRY1    DB 1, 2, 5, 7, 9, 10, 15, 19, 20, 25
    ARRY2    DB 40, 1, 0, 3, 4, 5, 6, 9, 10, 30
    INDT     DB FFH
DATA ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA, ES: EXTRA
START:    MOV AX, DATA
        MOV DS, AX
        MOV ES, AX
        LEA SI, ARRY1
        LEA DI, ARRY2
        CLD
        MOV CX, 10
;
;Compare the byte at DS:SI with ES:DI. Repeat if equal.
;
        REPE CMPSB
        JNE NEQ
        MOV INDT, 01H
        JMP FINI
NEQ:     MOV INDT, 00
FINI:    NOP
CODE ENDS
```

EXAMPLE 6.10

Convert a two-digit BCD number stored in memory into a hexadecimal number. Use the NEAR procedure call.

Solution:

```
DATA SEGMENT
BCDNO     DB      76H
HEXNO     DB      00H
DATA ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
START:    MOV AX, DATA
```

```

MOV DS, AX
MOV AL, BCDNO
CALL BCDCNV
MOV HEXNO, AL
NOP

BCDCNV      PROC NEAR
    MOV BL, AL
    AND AL, F0H
    MOV CL, 4
    ROR AL, CL
    AND BL, 0FH
    MOV BH, 10
    MUL BH
    ADD AL, BL
    RET

BCDCNV      ENDP
CODE ENDS

```

EXERCISES

1. Develop a program to find out the number of occurrences of a number 74H in an array of numbers stored in memory. Store the number of occurrences in memory location N-OCCUR.
2. Write a program to divide a 16-bit number by an 8-bit number.
3. Develop a program to add the elements of two 2 × 2 matrices in the memory.

6.11 MISCELLANEOUS INSTRUCTION GROUPS

Under this group, a number of instructions for modifying the bits of Flag Register have been included. The standard NOP and HLT are also present. The 8086 also provides three different types of interrupt instructions—INT, INTO and IRET. The INT instruction is basically used as a 2-byte call instruction. It is similar to the RST instruction of the 8085. The INT is normally used to execute the interrupt service routines required for the peripheral device. The second byte specifies which of the 256 possible interrupt servicing routines are to be executed. The INTO instruction causes the Overflow Flag to be checked, and ISR for Overflow Interrupt (whose locations are stored at memory location

010H) to be executed, if the Overflow Flag is set; otherwise the next instruction in sequence is executed. The IRET instruction is Return from Interrupt instruction and is the last instruction in any ISR.

CLC (1 byte)—Clear Carry

Flags affected: CF.

Clears the Carry Flag.

(CF) \square 0

CMC (1 byte)—Complement Carry Flag

Flags affected: CF.

Toggles (inverts) the Carry Flag.

(CF) \leftarrow $\overline{\text{CF}}$

STC (1 byte)—Set Carry

Flags affected: CF.

Sets the Carry Flag to 1.

(CF) \square 1

CLD (1 byte)—Clear Direction Flag

Flags affected: DF.

Clears the Direction Flag causing the string instructions to auto-increment the SI and DI index registers.

(DF) \square 0

STD (1 byte)—Set Direction Flag

Flags affected: DF.

Sets the Direction Flag to 1 causing the string instructions to auto-decrement SI and DI instead of auto-increment.

(DF) \square 1

CLI (1 byte)—Clear Interrupt Flag (disable)

Flags affected: IF.

Disables the maskable hardware interrupts by clearing the interrupt flag. NMIs and the software interrupts are not inhibited.

(IF) \square 0

STI (1 byte)—Set Interrupt Flag (Enable Interrupts)

Flags affected: IF.

Sets the Interrupt Flag to 1 which enables recognition of all hardware interrupts.

(IF) \square 1

FALC (1 byte)

Flags affected: None.

(AL) \square 0, if (CF) = 0, (AL) \square FF, if (CF) = 1

Fills AL with Carry.

HLT (1 byte)—Halt CPU

Flags affected: None.

Halts CPU until RESET line is activated, NMI or maskable interrupt received. The CPU becomes dormant but retains the current CS:IP for a later restart.

NOP—No Operation

Flags affected: None.

This is a do-nothing instruction. It results in the occupation of both space and time and is most useful for patching code segments. (This is the original XCHG AL, AL instruction.)

WAIT/FWAIT (1 byte)—Event Wait

Flags affected: None.

CPU enters wait state until the coprocessor signals that it has finished its operation. At that instant, TEST pin receives a high input signal. This instruction is used to prevent the CPU from accessing memory that may be temporarily in use by the coprocessor. WAIT and FWAIT are identical.

ESC DADDR (2, 3 or 4 bytes)—Escape

Flags affected: None.

Provides access to the data bus for other resident processors. The CPU treats it as a NOP, but places the contents of memory location DADDR on bus.

? \square (EA)

LOCK (1 byte)—Lock Bus

Flags affected: None.

This instruction is a prefix that causes the CPU assert bus lock signal during the execution of the next instruction. It is used to avoid two processors from updating the same data location. This should only be used to lock the bus prior to XCHG, MOV, IN and OUT instructions.

INT num—Interrupt

Flags affected: TF, IF.

Initiates a software interrupt by pushing the flags, clearing the Trap and Interrupt Flags, pushing CS followed by IP and loading CS:IP with the value found in the interrupt vector table. The execution, then begins at the location addressed by the new CS:IP.

INTO (1 byte)—Interrupt on Overflow

Flags affected: IF, TF.

If the Overflow Flag is set, this instruction generates an INT 4 which causes the code addressed by 0000:0010H to be executed.

IRET (1 byte)—Interrupt Return

Flags affected: AF, CF, DF, IF, PF, SF, TF, ZF.

Returns control to the point of interruption by popping IP, CS and then the Flags from the stack and continues the execution at this location. CPU exception interrupts will return to the instruction that caused the exception, because the CS:IP, placed on the stack during the interrupt, is the address of the offending instruction.

6.12 CONCLUSION

In the present chapter we have dealt with the the 8086 addressing modes, the specific assembler directives and the instruction set. Through various examples, we have also seen how program development is planned and done for applications involving the 8086 microprocessor. The power of the 8086 processor is evident through its powerful addressing modes and instruction sets.

In the next chapter we shall deal with the interfacing of both the 8085 and the 8086 microprocessors in various application environments.

EXERCISES

1. A two dimensional 3 \times 3 array of numbers is stored in memory starting from 2000H. Find whether a particular number 76H is present in the array. If yes, then the memory variables ROW and COLUMN should store the row number and the column number of the element, otherwise both variables should be zero.
2. Find the maximum number in an array of 10 numbers. Store the number in MAX and location from start of the array in LOC. Both MAX and LOC are memory locations.
3. Two numbers C1 and C2 are stored in memory. The operation between these numbers is dictated by the bits 5 and 6 of another number C3 as follows:

If bit 5 of C3 = 1 and bit 6 = 0, then

$$C1 = (C1 \text{ AND } C2) \text{ OR } F0H$$

$$C2 = (C1 \text{ XOR } C2) \text{ AND } F0H$$

If bit 5 = 0 and bit 6 = 0, then

$$C1 = (C1 \text{ AND } 07H) \text{ OR } C2$$

$$C2 = (C2 \text{ OR } 90H) \text{ XOR } 50H$$

Otherwise—no operation.

Develop a program.

4. In a factory, the wages of workers are paid at the end of the week. To calculate the wages, a matrix consisting of the worker code number, the age and the number of hours worked every day, i.e. day 1 (Monday) to day 7 (Sunday), is maintained. Thus, a matrix having 10 columns is created. The number of rows will be the number of workers employed. The last column will be the total wages payable for the work in the week.

The wages are paid in three slabs:—wage1 for those who are less than 20 years, wage2 for those who are 20 years or more than 20 years but less than 40 years, and wage3 for those who are 40 years or more than 40 years.

Considering that the factory has 20 workers, develop a program to calculate the wages payable to workers at the end of the week, and the total wages payable in the week (column 10). Assume that the matrix is already stored in memory with all the data, and wage1, wage2 and wage3 are stored in memory.

5. In a post office, the workers sort out the letters based on pin code numbers. *Consider that the first three digits of pin code identify the main post office of destination.* Out of these, the first digit identifies the region, the second digit identifies the sub-region, and the third digit identifies the sub-sub-region of the destination. Let us assume that a letter travels from a region to its sub-region, to its sub-sub-region and then finally to its post office of destination. (This may not be true in real sense.) Assume 10 regions, each having 10 sub-regions, and each sub-region having 10 sub-sub-regions.

Develop a program to computerize the sorting process in region, sub-region and sub-sub-regions. Thus, when you enter a pin number using the keyboard (or the program may take from memory), the region slot is activated followed by a particular sub-region and then the sub-sub-region slot is activated. At the sub-sub-region the number of letters along with the pin number is maintained.

It is important that the application is planned, documented and then the program is developed and tested.

FURTHER READING

- Gilmore, Charles M., *Microprocessors Principles and Applications*, McGraw-Hill, 1989.
- Hall, Douglas V., *Microprocessors and Interfacing Programming and Hardware*, Tata McGraw-Hill, 2003.
- Intel Data Book on Intel 8086 Microprocessor*, Intel Corporation, Santa Clara, California.
- Osborne, A. and Kane, J., *An Introduction to Microprocessors—Some Real Microprocessors*, Osborne & Associates Inc., 1978.

MICROPROCESSOR PERIPHERAL INTERFACING

7.1 INTRODUCTION

A microprocessor-based system will contain memory, an input unit and an output unit, apart from the microprocessor itself. We discussed the memory interfacing of microprocessor in the previous chapters. We shall discuss in this chapter interfacing of input–output peripherals with microprocessors. The data to microprocessor may come through a keyboard, or through another system in the form of digital inputs or in the form of analog inputs from sensors, etc.

The output devices may be a seven-segment LED display, a CRT, or a printer. It can also be a control valve, which is controlled through analog voltage. The present chapter is dedicated to all the aspects of interfacing microprocessors to different peripherals. Even though the design and the software are described for the 8085 and the 8086 microprocessors, the concepts are general enough and can be applied to any other microprocessor.

There are a variety of support IC chips for peripheral interfacing of the 8085 and the 8086 microprocessors. We shall describe in brief only the following support chips, as these are widely used.

- (a) Programmable peripheral interface Intel 8255.
- (b) Keyboard and display controller Intel 8279.
- (c) Programmable interval timer Intel 8253/8254.

However, other IC chips like DMA Controller and Interrupt Controller have not been covered. Similar support chips of other manufacturers are available for their microprocessors. Only the operational aspects of the chips mentioned at (a), (b) and (c) are presented here. The user is advised to refer to the user manual before using any of these chips in any circuit.

7.2 GENERATION OF I/O PORTS

While discussing the interfacing of the microprocessor in Chapters 2, 3 and 5, we observed that the microprocessor inputs and outputs the data through ports. A port is a slot where one unit of data, e.g. a byte or a word can be stored or can be read. Microprocessors access the ports through their unique addresses called port numbers. These addresses are output on the address lines, decoded by the address decoder, and the chip select (\overline{CS}) signal is generated for the port.

The simplest form of decoding is the linear select method as shown in Figure 7.1 for the 8085. This requires the smallest amount of logic but can only be used for a small number of input and/or output ports. One address bit is associated with each I/O port and is logically ANDed with $\overline{IO/M}$ and \overline{RD} or \overline{WR} to generate an input or output device select pulse. For the 8086, the signal “ $\text{NOT}(M/\overline{IO})$ ” will be used in place of $\overline{IO/M}$ of the 8085.

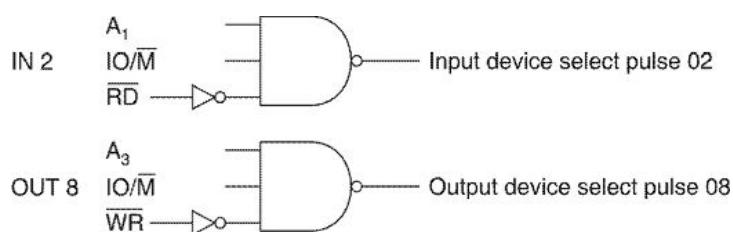


Figure 7.1 Generation of single device select pulse.

However, if the number of ports is large, then the 8205 (1 out of 8) decoder chip of Intel (Figure 7.2) or a number of 74139 decoders can be used (Figure 7.3) to generate input and output device select pulses. Decoders can also be cascaded to generate a large number of device select pulses. Figure 7.4 shows the number of 74154 (1 out of 16) decoder chips used to generate a total of 256 device select pulses. It must be noted that the 8085 facilitates 256 input and 256 output ports.

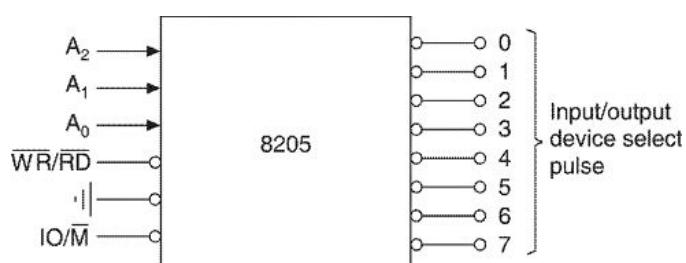


Figure 7.2 Generation of multiple device select pulse using the 8205.

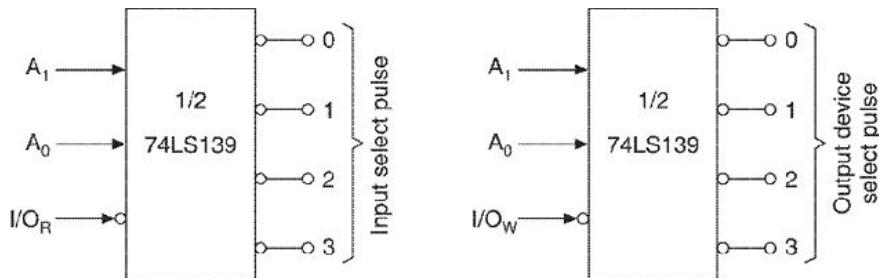


Figure 7.3 Generation of multiple device select pulse using the 74LS139 decoder.

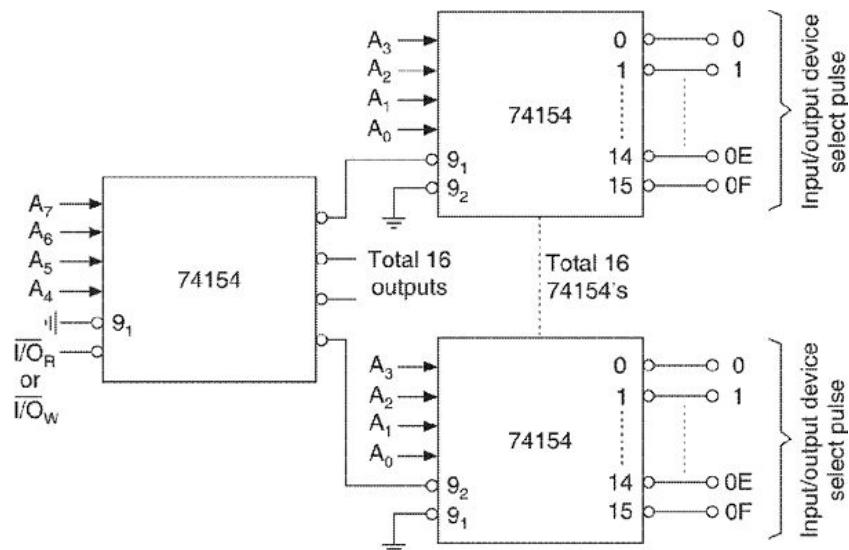


Figure 7.4 Generation of 256 input-output device select pulses using 74154.

The I/O ports can be implemented with SSI, MSI, or LSI circuits. The 8282/8283 chips of Intel can be used as I/O ports. These have been described in Chapter 5, and can be configured for various applications.

EXAMPLE 7.1

Design an example to illustrate the use of a microprocessor in a weigh balance system.

Solution:

A microprocessor in an automatic weigh balance will require two ports—one input port to accept the weight information and one output port to display the information.

Let us assign the port no. 4 to input port and the port no. 8 to output port. Figure 7.5 shows the basic circuit for 8-bit input and output operation in this application. The ports in this case are simple registers connected to weighing and display circuits.

We know that in the 8085 the data is transferred at the falling edge of \overline{RD} or \overline{WR} signal. Thus, at that instant the data must be in the data bus in case of read operation. If you closely analyze, you will find that the circuit is able to carry out the task. In case of minimum mode of the

8086, the signal IO/M will be replaced by M/IO . The data bus is 16 bits and thus 16-bit ports will replace 8-bit ports (Figure 7.6).

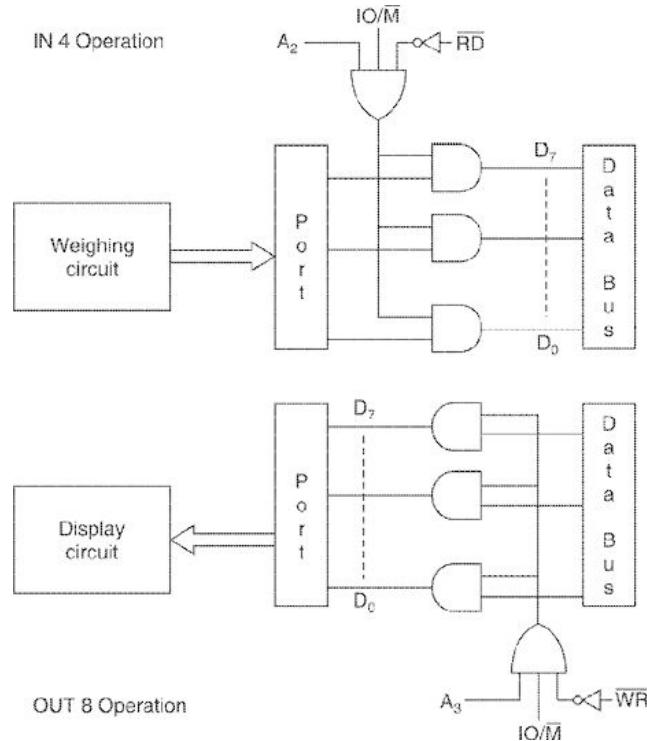


Figure 7.5 Input–output port generation for the weighing machine example for the 8-bit systems.

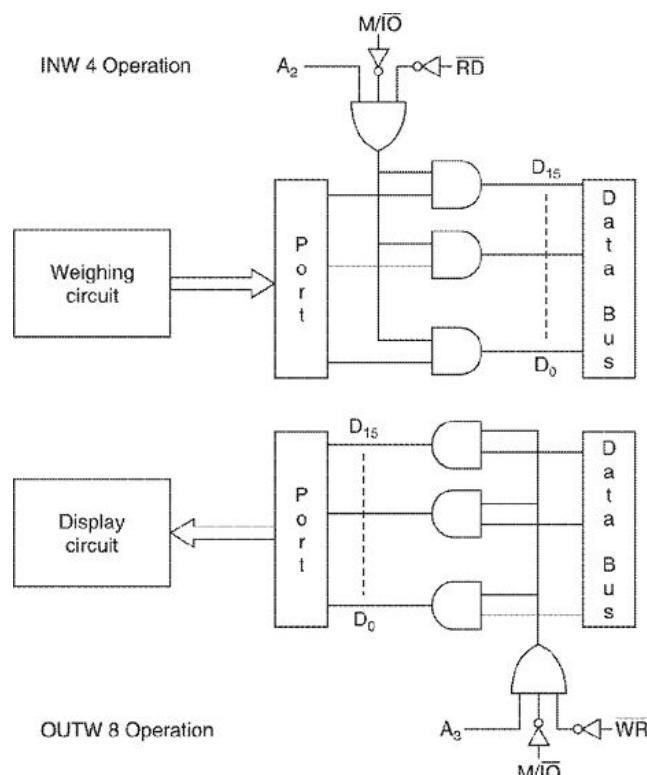


Figure 7.6 Input-output port generation for the weighing machine example for the 16-bit systems.

EXERCISES

1. Modify the circuit in Figure 7.5 for input port no. 7 and output port no. 10.
2. Modify the circuit in Figure 7.6 for maximum mode signals of the 8086.

7.3 PROGRAMMABLE PERIPHERAL INTERFACE (PPI)—INTEL 8255

A port can be the input port (i.e. the microprocessor can read the data from the port) or the output port (i.e. the microprocessor will output the data to the port). Though it is technically possible for any port to be both input and output ports, it is not recommended or done as a measure of good design practice. It makes the circuitry complicated and creates confusion at times. Some input–output devices require handshake protocol for data transfer while some others may require interrupt driven data transfer. Thus, every time one needs to interface a device to the microprocessor, special circuit development may be necessary.

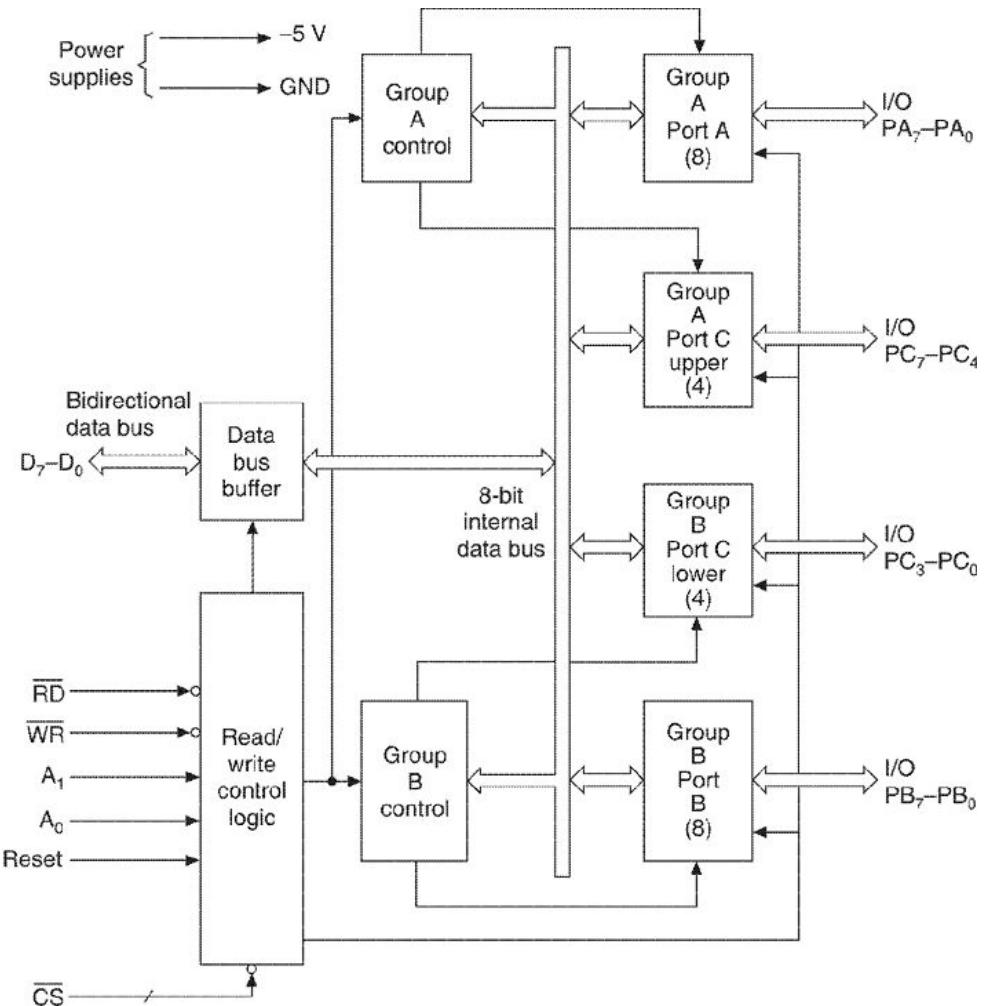


Figure 7.7 The 8255A block diagram.

This problem was foreseen by the microprocessor manufacturers, and hence specialized IC chips were designed and manufactured. These chips are called Programmable Peripheral Interface (PPI) and as the name suggests, they can be programmed to interface various peripherals to the microprocessors. The 8255 PPI is widely used with microprocessors for peripheral interfacing.

Figure 7.7 shows the internal block diagram of the 8255 PPI. It has 24 I/O pins distributed to four ports—port A, port B, port C (upper) and port C (lower). These ports are divided into two groups. Group A has port A and port C (upper) and group B has port B and port C (lower). While port A and port B have eight I/O lines each, port C (upper) and port C (lower) have four I/O lines each, thus accounting for 24 I/O lines. D₇–D₀ account for data bus, \overline{RD} and \overline{WR} for read and write control signals, \overline{CS} for chip select from address decoder and A₀, A₁ for port address. It is a 40-pin IC in dual-in-line package.

Each port can be programmed as input or output. The microprocessor outputs a control word to the 8255. The control word contains information that initializes the functional configuration of the 8255. There are three basic modes of operation that can be selected by the software. These modes are described below.

7.3.1 Mode 0—Basic Input/Output

This mode provides simple input and output operations for each of the three ports. Data is simply written to or read from a specified port.

Basic functional definition

In mode 0:

- There are two 8-bit ports (A and B) and two 4-bit ports (C (lower)) and (C (upper)).
- Any port can be an input port or an output port.
- Outputs are latched.
- Inputs are not latched.
- 16 different input/output configurations are possible in this mode.

7.3.2 Mode 1—Strobed Input/Output

It provides means for transferring I/O data to or from a specified port in conjunction with strobes or handshaking signals. Port A and port B use the lines on port C for handshaking signals.

Basic functional definition

In mode 1:

- There are two groups (group A and B).
- Each group contains one 8-bit data port and one 4-bit control data port.
- The 8-bit data port can be either an input port or an output port. Both inputs and outputs are latched.
- The 4-bit port is used for control as well as for status of the 8-bit data port.

7.3.3 Mode 2—Strobed Bidirectional Bus

This functional configuration provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data. Handshaking signals are provided to maintain a proper bus flow discipline. Interrupt generation and enable/disable functions are also available.

Basic functional definition

In mode 2 (used in Group A only):

- There is one 8-bit bidirectional bus port (port A) and a 5-bit control port (port C)
- Both inputs and outputs are latched.
- The 5-bit control port (port C) is used for control as well as for status of the 8-bit bidirectional bus port (port A).

7.3.4 Configuring the 8255

The 8255 can be configured to work in one of the above modes by the microprocessor through a control word. This control word is sent by the microprocessor to the 8255. Ports A, B, C, and the control register port are addressed by A₀, A₁ pins as follows:

A ₁	A ₀	Port
0	0	A
0	1	B
1	0	C
1	1	Control register

While both the input and output operations are possible on A, B and C ports, the control register port can only be written into. No read operation of the control register is allowed.

Figure 7.8 shows the control word format. As an example, if we want two 8-bit output ports, and one 4-bit input port for the interfacing of a device to microprocessor in mode 0, then the control word will be

$$1\ 0\ 0\ 0\ 0\ 0\ 0\ 1 = 81H$$

This makes ports A, B, C (upper) as output and port C (lower) as input.

The mode is 0, i.e. basic input/output. Now, the 8085 and the 8086 microprocessors can write/read data by using OUT/IN instructions.

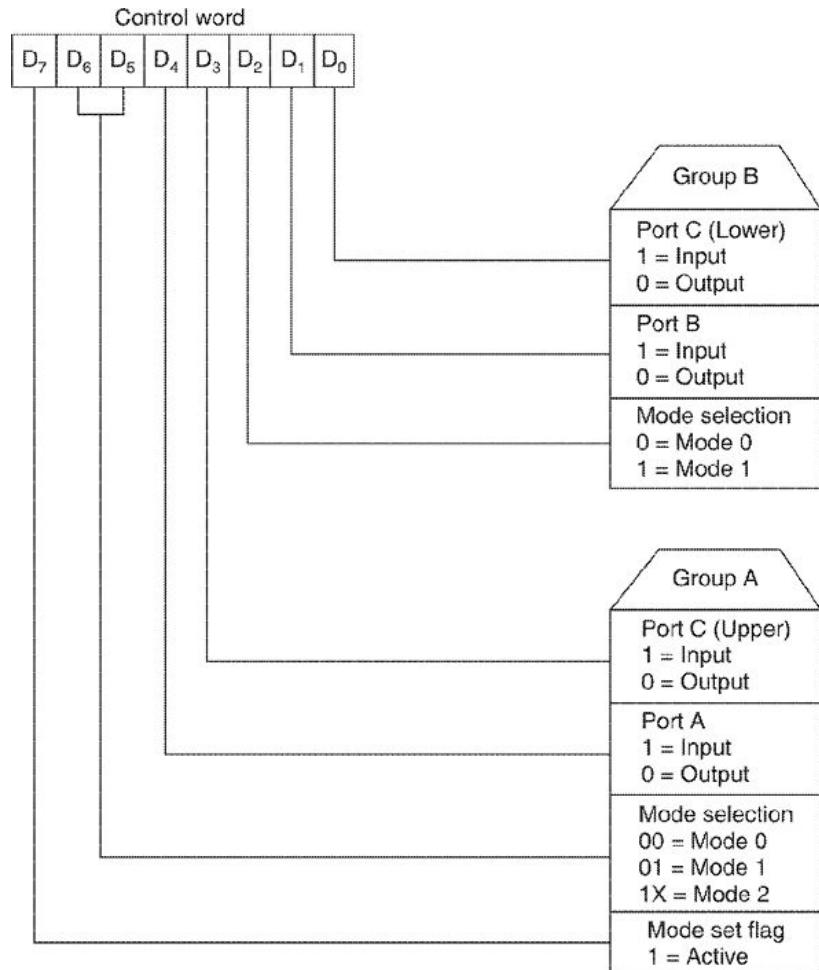


Figure 7.8 Control word format of the 8255 PPI.

7.3.5 Interfacing the 8255

Figure 7.9 shows the interfacing of the 8255 with the 8085 microprocessor. The address decoder generates the chip select signal; \overline{RD} and \overline{WR} are directly connected to the 8085 read/write control signals, and the RESET is connected to the general reset. In the present interfacing, the port numbers for A, B, C and control register ports are 0, 1, 2 and 3 respectively. The software instructions IN (Port No.) and OUT (Port No.) are used for data transfer.

The interfacing of the 8255 with the 8086 will require the use of two Intel 8255 chips for 16-bit input-output.

In case only the 8-bit input-output is desired using AD₀ to AD₇ lines, only one Intel 8255 will then be required and the circuit (in minimum mode) will be almost similar to the one shown in Figure 7.9. The signal M/IO will be inverted before connecting to the decoder. Figure 7.10 shows the interfacing of the 8086 in the minimum mode to the 8255 for the 16-bit input-output operation.

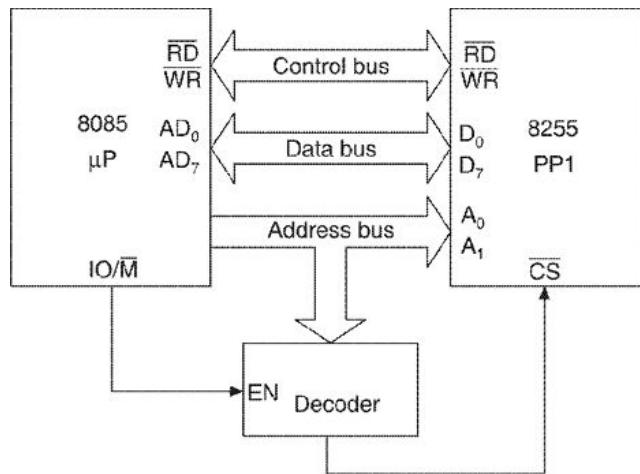


Figure 7.9 Interfacing the 8255 PPI to the 8085 microprocessor.

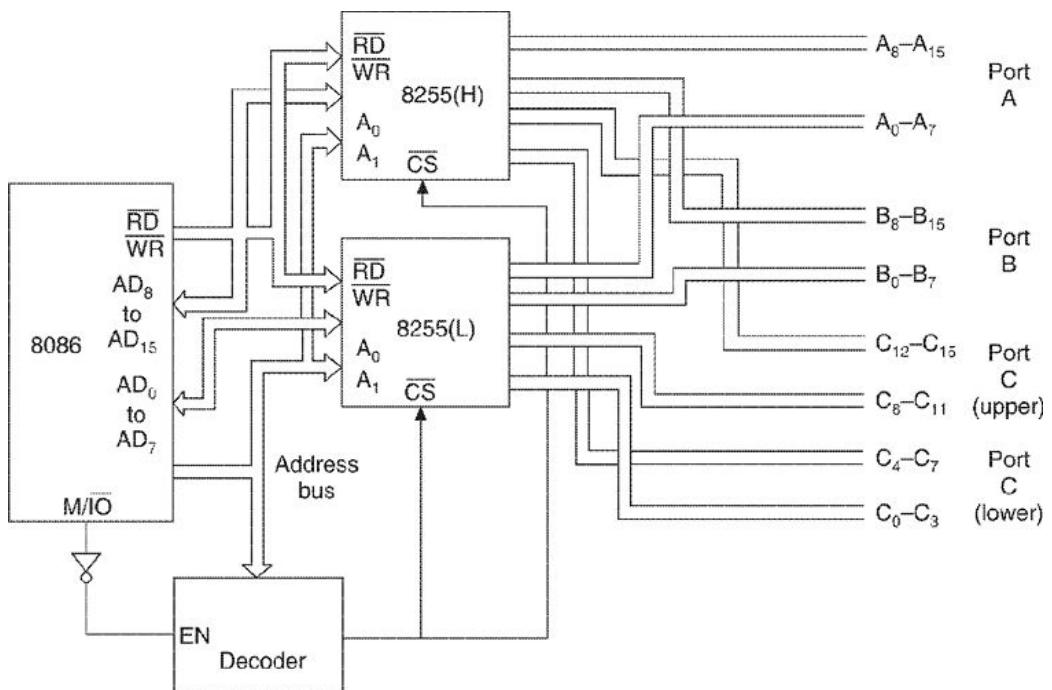


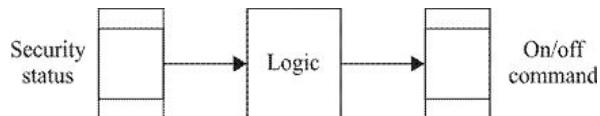
Figure 7.10 Interfacing of the 8255 PPI to the 8086 (minimum mode) for the 16-bit input-output operation.

EXAMPLE 7.2

Illustrate the use of the microprocessor that accepts inputs from eight different systems about their security settings (1 = On, 0 = Off), and then based on a well-defined logic decides the systems which must be switched On or Off (1 = On, 0 = Off).

Solution:

The input may be assembled in one byte, each bit representing the security status of one system.



Similarly, the On/Off command may be represented in one byte, each bit representing the command for one system.

Using the 8085 microprocessor

Figure 7.11 shows the circuit diagram using the 8085 interfaced to the 8255. Port A has been used as the input port for the security status, whereas port B has been used as the output port for the command. The port numbers assigned are 04 (port A), 05 (port B), 06 (port C) and 07 (Control Word) as evident from the circuit.

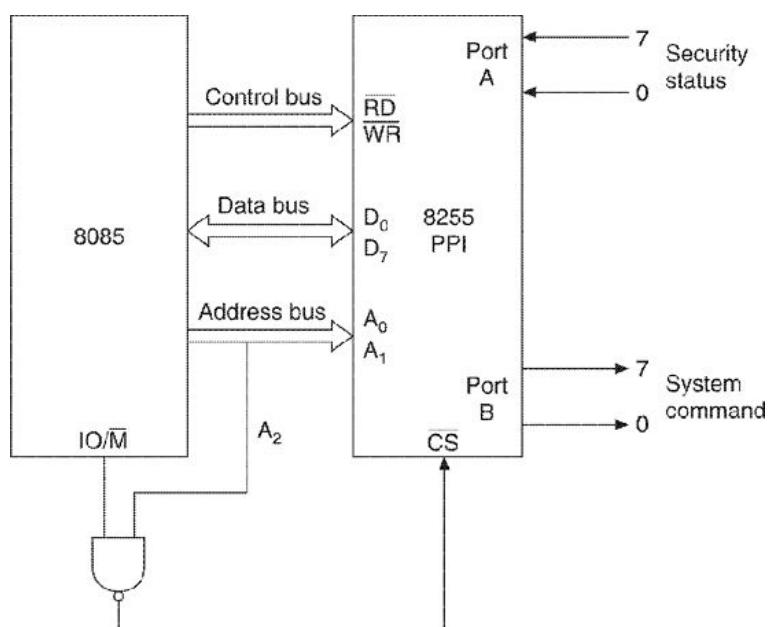


Figure 7.11 The 8085-based system (circuit diagram for the system security Example 7.2).

The operations will be as follows:

(i) Configure the 8255

A = Input port, B = Output port, Mode = 0

Control word = 1 0 0 1 0 0 0 0
--

= 90H

MVI A, 90H
OUT 07H

(ii) Input Security Status

IN 04H

(iii) Logic

(iv) Output command

OUT 05H

Using the 8086 microprocessor

In case of the 8086, the operation will require only 8-bit input-output in the minimum mode. The circuit is shown in Figure 7.12. Following are the port assignments:

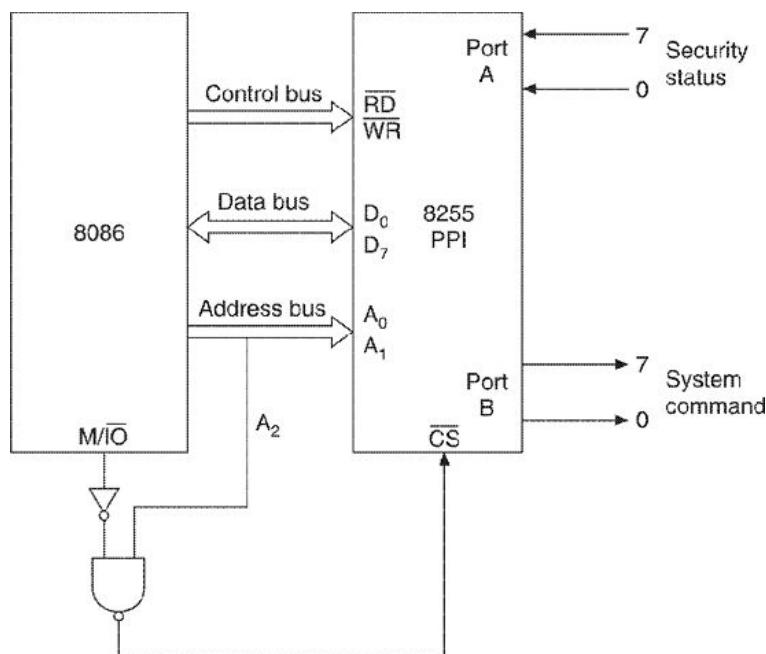


Figure 7.12 The 8086-based system (circuit diagram for the system security Example 7.2).

Port	Port address	Signal
A	04	Input Security Status
B	05	Output Command
C	06	Output (not used)
Command Word	07	Output

The control word for mode 0 will be:

1 0 0 1 0 0 0 = 90H

The operations will be:

(i) Configure Intel 8255

MOV AL, 90H

OUT 07H, AL

(ii) Input Security Status

IN AL, 04H

(iii) Logic

—
—
—

(iv) Output Command

OUT 05H, AL

EXERCISES

1. Redraw the circuits (Figure 7.11 for the 8085 and Figure 7.12 for the 8086) assigning port no. 20 decimal to port A, and so on.

Write the program for the following logic and test.

security status = SS

system command = SC

SC = (SS AND EFH) OR (\overline{ss} AND F0H) OR (SS AND 07H)

(You may also select various other logics and execute.)

2. Redraw the circuit in Figure 7.12 for 16-bit input–output and write the program using the logic of Exercise 1 above.

7.4 SAMPLE-AND-HOLD CIRCUIT AND MULTIPLEXER

The multiplexer and the sample-and-hold (S&H) circuit are normally used together in a data acquisition system. The S&H circuit may be connected to the multiplexer in two configurations. The choice of a particular configuration is dictated by the application.

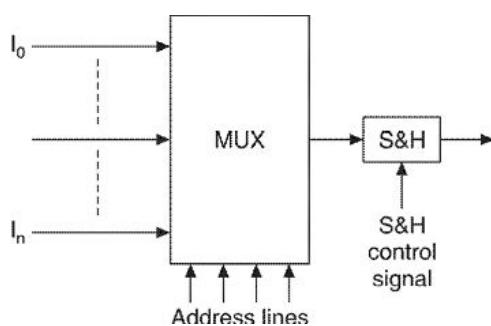


Figure 7.13 Sample-and-hold circuit placed after the multiplexer.

Figure 7.13 shows the configuration in which the sample-and-hold circuit is used after the multiplexer. The microprocessor selects one of the input channels through address lines. When the S&H control signal is low (i.e. Sample signal), the S&H circuit tracks the analog input. When

the S&H control signal becomes high (Hold signal), the analog input is latched.

The multiplexer outputs the analog value on that channel and the S&H circuit latches it when the hold signal is received from the microprocessor. This value remains latched until another signal is issued to S&H circuit by the microprocessor. In this configuration, the values at all the channels cannot be obtained at the same time. Thus, for the processes where the signal is changing slowly or for the processes where all the channels are to be sampled sequentially, this configuration will be useful.

However, there are processes in which the signals change at a faster rate and all the channels need to be sampled at the same time. These processes require that each channel should have a separate S&H circuit, resulting in the configuration shown in Figure 7.14. The microprocessor issues the hold signal for all the channels at the same time. The analog values present on the channels are latched in a S&H circuit. The microprocessor then selects the channels and the analog values are taken for further processing.

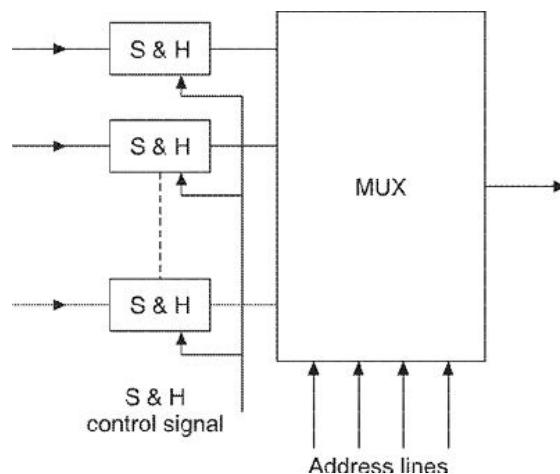


Figure 7.14 Multiplexer placed after the S&H circuit connections.

The interface of the microprocessor to S&H circuit and multiplexer requires two types of signals. The S&H circuit requires one signal for the S&H command. The multiplexer will require address signals to come from the microprocessor. For an 8-channel multiplexer, three address lines will be required. These signals can be generated by software through the I/O port. The I/O ports are selected by their port numbers under software control. Figures 7.15 and 7.16 show the interfacing of the microprocessor with a S&H circuit and a multiplexer.

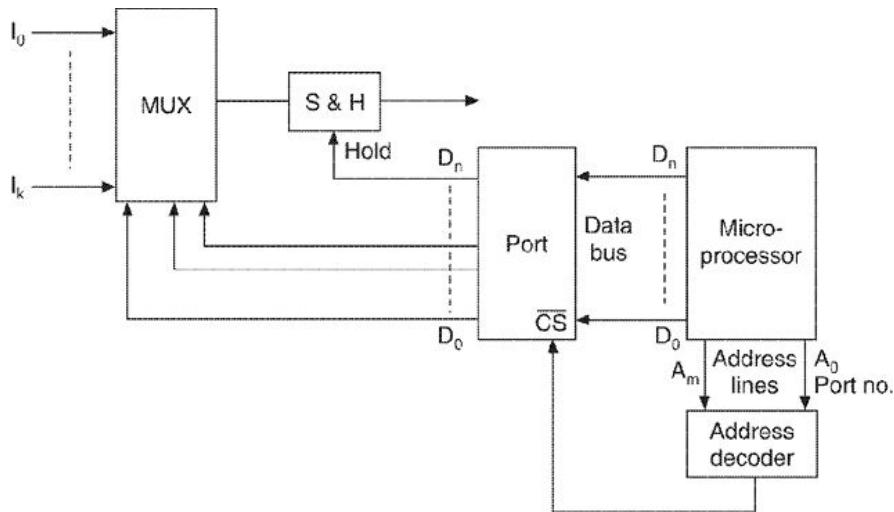


Figure 7.15 Interface with microprocessor of S&H circuit placed after the multiplexer.

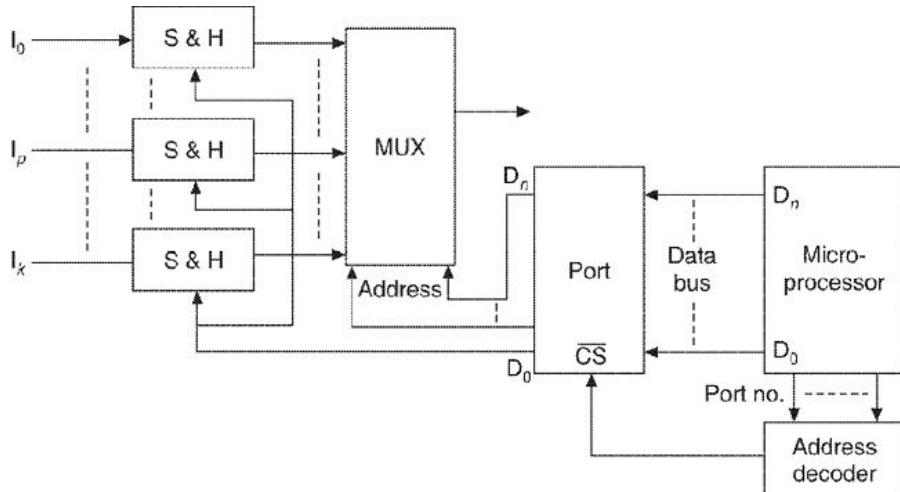


Figure 7.16 Interface with microprocessor of multiplexer after the S&H circuit connection.

EXAMPLE 7.3

Extend the design of Example 7.2 to include the security status of each system on one of the lines available as system output. The status of these lines needs to be read and then the system command decided based on pre-defined logic.

Solution:

The system status lines may be input to the multiplexer which is interfaced to the microprocessor through the 8255. In addition, the S&H circuit will also be required to hold the signal to the constant value. The following input-output signals are required from the 8255.

Channel Select	—	Output 3 lines—Port C (L)
Hold	—	Output 1 line—Port C (U)—line 0
Output of S&H	—	Input 1 line—Port B—line 0

Command to System — Output 8 lines, one for each system—
Port A

The 8255 configuration—Port A = Output, Port B = Input, Port C = Output, Mode = 00

Control word:

1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

$$= 82 \text{ H}$$

Let us solve this using first the 8085 microprocessor and then the 8086 microprocessor.

Using the 8085

Figure 7.17 shows the circuit schematic of the 8085 processor with the 8255 PPI along with the S&H and multiplexer.

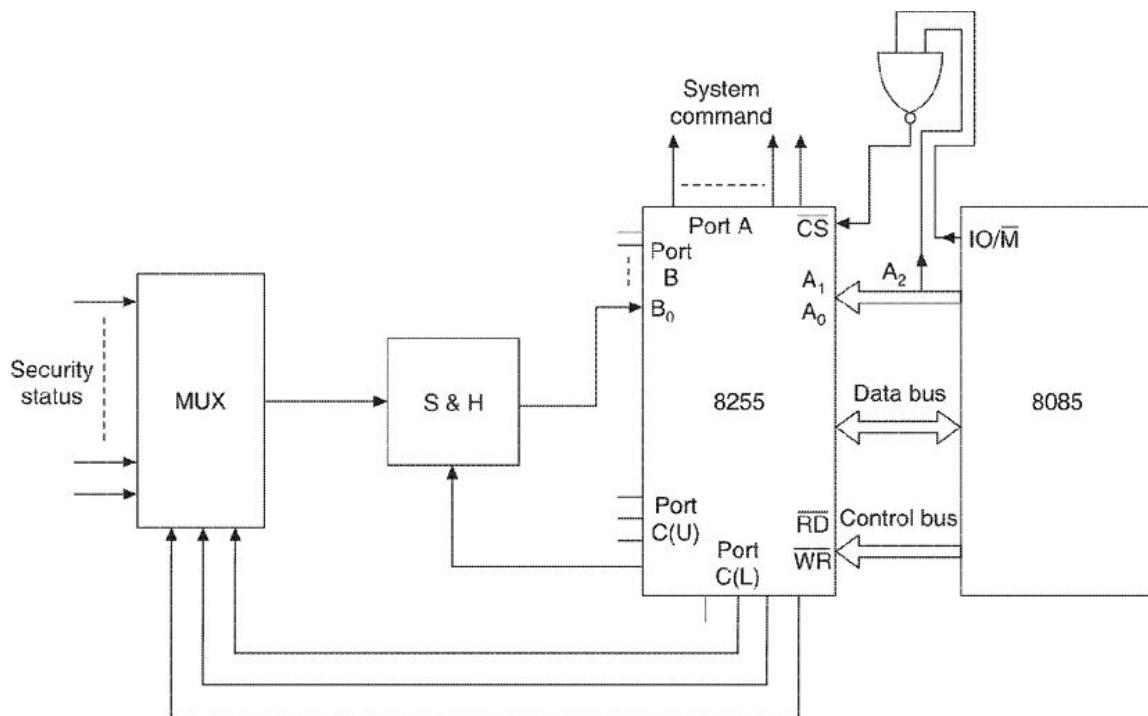


Figure 7.17 The 8085 schematic diagram for system security using the S&H circuit.

Let us now develop the software.

MVI A, 82H

OUT 07H

; Select channel 1 and issue the hold command.

; Port C =

0	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---

MVI A, 10H

```
OUT    06H  
; Read the security status.
```

```
MVI    A, 00H  
IN     05H
```

; Logic

—
—
—

```
; System command byte stored in C register  
; Issue the system command.
```

```
MOV    A, C  
OUT    04H
```

Using the 8086

The circuit schematic using the 8086 is shown in Figure 7.18. The S&H circuit is placed after the multiplexer and both are interfaced to the microprocessor using the 8255 PPI.

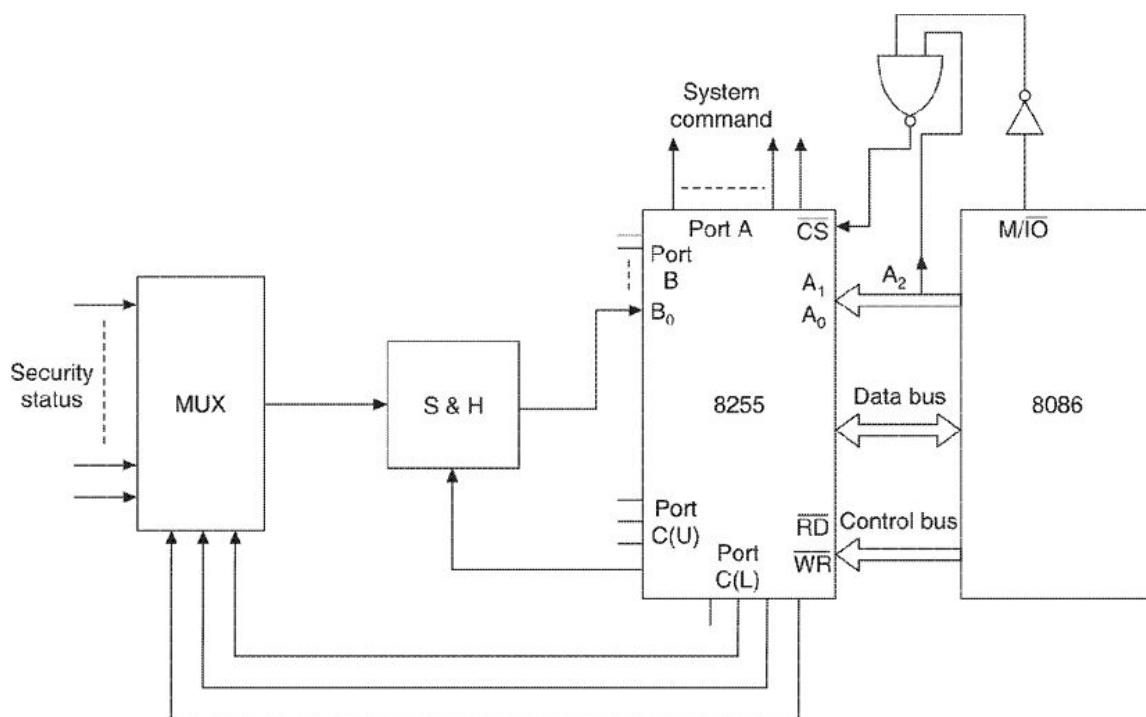


Figure 7.18 The 8086 schematic diagram for system security using a S&H circuit.

Let us now develop the software.

```
DATA      SEGMENT  
          COMND DB      0  
DATA      ENDS
```

```

CODE      SEGMENT
          ASSUME CS: CODE, DS: DATA
MOV       AX, DATA
MOV       DS, AX
; Initialize Intel 8255
MOV       AL, 82H
OUT      07H, AL
; Select channel 1 and issue the hold command.
MOV       AL, 10H
OUT      06H, AL
; Read the security status.
MOV       AL, 00H
IN        AL, 05H
; Logic
; Determine the system command and store in COMND.
—
—
—
; Issue the system command.
MOV       AL, COMND
OUT      04H, AL
NOP
CODE    ENDS

```

EXERCISES

1. In Example 7.3, the S&H circuit is placed after the multiplexer. Modify the circuit to implement the scheme in which the multiplexer is placed after the S&H circuit. You may include any logic of your choice, for generating the system command. Write and test the software.
2. In the given design, an analog multiplexer has been used to accept the signal of the system security status. Can we use a digital multiplexer instead of the analog multiplexer? Study the working of the digital multiplexer. Design the circuit using a digital multiplexer, and write and test the software.

7.5 KEYBOARD AND DISPLAY INTERFACE

We shall now discuss the interfacing of keyboard and LED display with the microprocessor. We shall also deal with the interfacing of seven-segment LED displays both in serial and parallel modes.

7.5.1 Keyboard

A keyboard consists of number of key switches used for entering data, event, etc. The important part of the interface between the keyboard and the microprocessor is to find out whether any key has been pressed; if yes, then the location of the key should be determined. A key, when pressed, makes the contact till it is kept pressed and the contact is broken when it is released.

A key closure can be determined easily by supplying a voltage at one of the two points and finding out the voltage at the other point. For interfacing to microprocessor, one point is connected to +5 V and the other point is connected to sense the voltage (Figure 7.19).

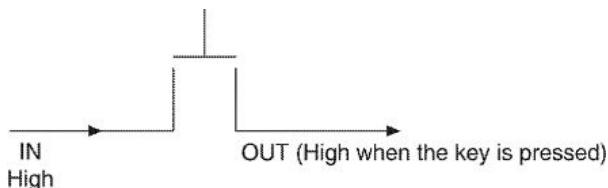


Figure 7.19 Basic keyboard operation—single key.

The OUT point can be sensed as one bit of any port. In case of two key switches, the input points of the two keys can be connected together and a high voltage applied. The OUT1 and OUT2 (Figure 7.20) points can be connected to two separate bits of a port.

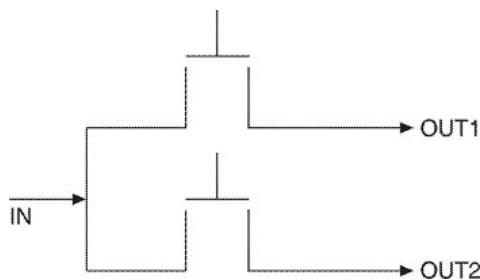


Figure 7.20 Basic keyboard operation—two keys.

Let us now consider the keyboard with four key switches. Extending the approach of Figure 7.20, we can interface this by connecting all the input points together and applying a high signal. The output points are connected to a port (Figure 7.21).

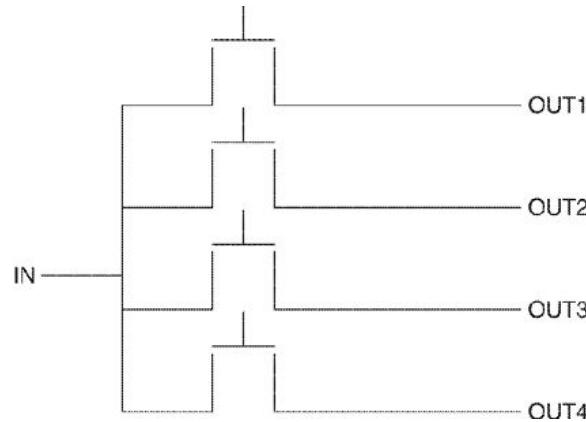


Figure 7.21 Basic keyboard operation—four keys.

It is evident that this approach cannot be adopted for more number of keys. As an example, a 64-key keyboard will require eight 8-bit ports for the output points. To solve such problems, the keyboard is arranged as a matrix consisting of rows and columns. For a 4-key keyboard, Figure 7.22 shows two rows IN1 and IN2 and two columns OUT1 and OUT2. The rows are energized sequentially and the voltages in columns are sensed. In the example of Figure 7.22, if IN1 is high and OUT2 is high, then the key 2 of the first row has been pressed. If IN2 and OUT2 are high, then the key 2 of the second row has been pressed.

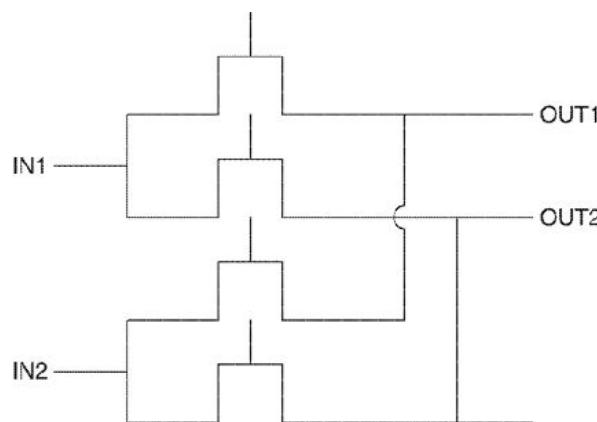


Figure 7.22 2 x 2 keyboard operation.

Thus, a 64-key keyboard can be arranged as a matrix with 8 rows and 8 columns. The interfacing will require two 8-bit ports for input and output lines. Let us assume that port 1 is connected to rows and port 2 to columns. Then the sequence of the operation is as follows:

- (i) Output row enable signal on port 1 for the first row.
- (ii) Input port 2 to check for key depression.
- (iii) Determine the column number of the depressed key in the keyboard.

(iv) Determine the code of the key and store it in memory.

The sequence from (i) to (iv) is repeated, for all the rows in a sequence, by the microprocessor. Figure 7.23 shows an 8 × 8 keyboard interface to the microprocessor.

To determine the column number of the depressed key in the keyboard, we would have to detect the bit in port 2 which is 1 and its relative position with respect to the bit corresponding to the first column. If bit 7 of port 2 corresponds to the first column, the bit 6 will then correspond to the second column and the bit 0 will correspond to the eighth column. This can be detected either by setting a mask pattern or by rotating.

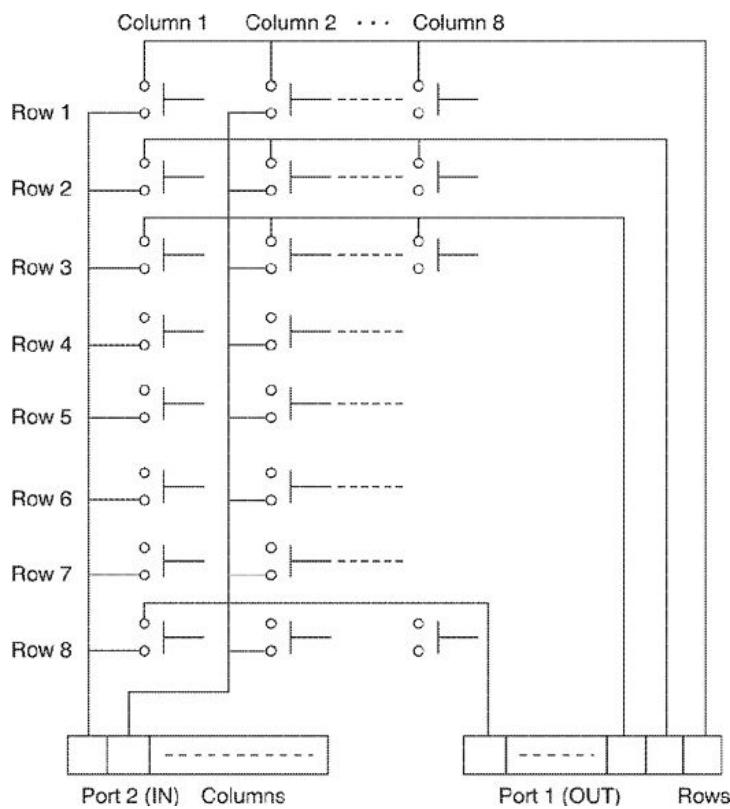


Figure 7.23 8 × 8 keyboard interfacing with microprocessor.

A table consisting of the codes of various keys in the keyboard is stored in the microprocessor memory. Whenever a key is pressed, its code is taken from the code table and used in the program. For easy access, the code table is organized in the manner shown in Figure 7.24. The codes for the keys for rows 1 to 8 are stored sequentially. Within a row, the codes for

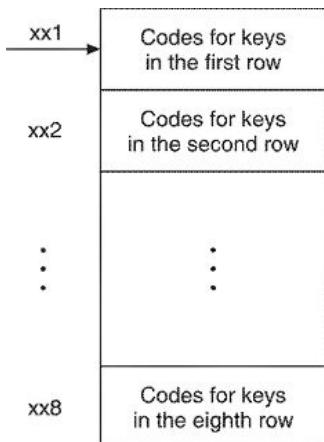


Figure 7.24 Code table for 8 × 8 keyboard.

the columns are stored sequentially. The addresses xx1, xx2, ..., xx8 are the starting addresses for the codes for the keys in the first, second ,..., eighth rows. Thus, if a key whose row number is 7 and the column number is 1 is depressed, the code of that key can be found at the address xx7. The program flowchart for the interface is shown in Figure 7.25.

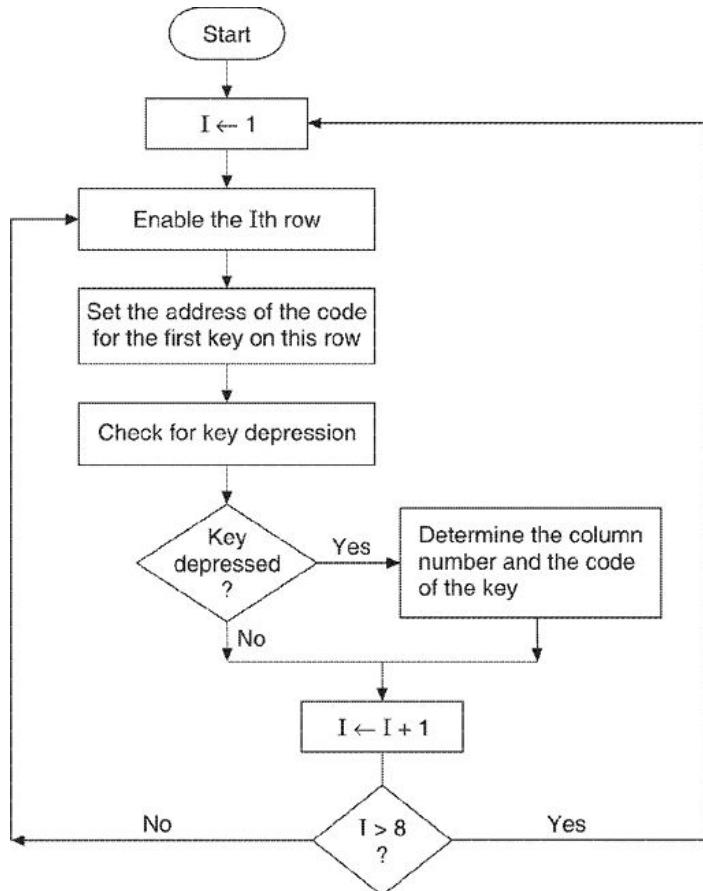


Figure 7.25 Keyboard—microprocessor interface software flowchart.

Key debouncing

When a key is depressed and released, the contact is not broken permanently. In fact, the key makes and breaks the contacts several times for a few milliseconds before the contact is broken permanently.

If a key depression is detected by a microprocessor, it is possible that the depression may be false, i.e. it may be due to the bouncing of the key. It is, thus necessary to debounce the key after depression. Debouncing by both hardware and software is possible. Hardware debouncing involves a circuit which ensures that the false depressions do not affect the output signal.

In software debouncing, the microprocessor executes a delay routine for a few milliseconds after the key depression is detected. The same key is again checked for depression. Further action like finding the code etc. is taken, only if the key is found depressed.

7.5.2 Light Emitting Diode Display

The light emitting diodes (LEDs) are used to display limited data, results, events, or messages. The total number of characters that are displayed is fixed, unlike the CRT display where any amount of data may be displayed on the screen. The LEDs are commonly used on the front panel of instruments, digital clocks, etc.

In most simple terms, the light emitting diode is a diode which when conducts, emits light energy. The conduction starts when the anode is held at a higher voltage than the cathode (Figure 7.26).

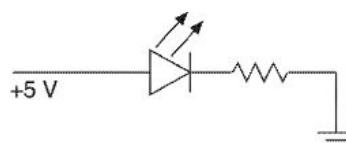


Figure 7.26 LED operation.

LEDs can be directly interfaced with the microprocessor through an output port, as shown in Figures 7.27 and 7.28.

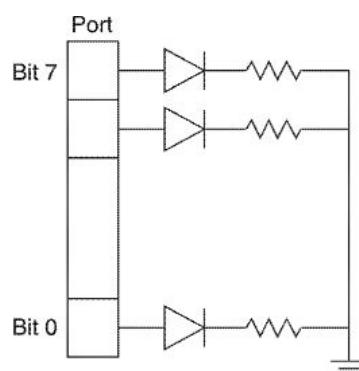


Figure 7.27 Microprocessor interface to LED (common cathode).

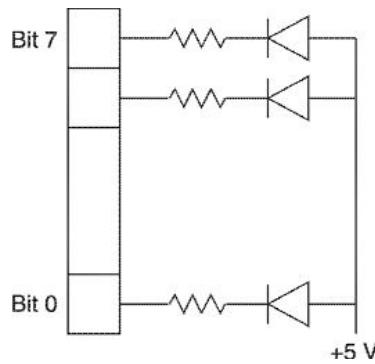


Figure 7.28 Microprocessor interface to LED (common anode).

For the interface shown in Figure 7.27, whenever a bit is 1, the corresponding LED will glow. In case of the interface shown in Figure 7.28, a 0 in any bit position will make the corresponding LED glow. So, by loading a particular bit pattern in the port, the desired LEDs may be lighted.

7.5.3 Seven-Segment LED

The LEDs can be arranged in the fashion shown in Figure 7.29.

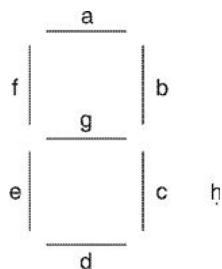


Figure 7.29 Seven-segment LED.

This structure has eight segments marked a, b, c, d, e, f, g, and h and is very useful in the display of the numeric and alphanumeric data. For example, to display character A, segments a, b, c, e, f and g should glow and to display character 6, segments a, c, d, e, f and g should glow (Figure 7.30). Originally, the structure had only seven segments a to g, and the eighth segment h (to display the decimal point) was added subsequently. Therefore, even though it has eight segments, the name seven-segment LED has been retained. Each segment may have more than one LED.

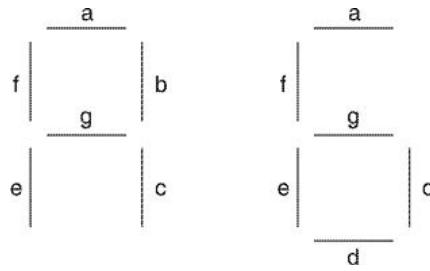


Figure 7.30 Character formation in seven-segment LEDs.

The On/Off information for eight segments can be arranged in one byte. The bit information can be connected to respective segments in the manner shown in Figure 7.28. Normally the LED anodes are connected to 5 V permanently and the cathodes are connected to bits as shown in Figure 7.28. Figure 7.31 shows the information for different segments.

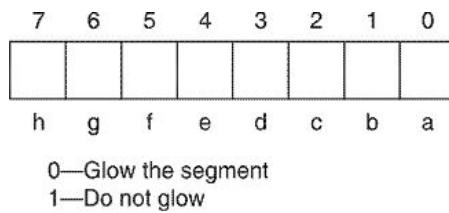


Figure 7.31 Control byte for seven-segment LEDs.

Parallel interface

The parallel interface between the seven-segment LED display and the microprocessor is shown in Figure 7.32. The anodes of all LEDs are held permanently at +5 V, whereas the cathodes are connected to the port bits. The microprocessor will load any bit pattern in the Out port. Those segments, for which 0 is stored in the bit pattern of the code, will be lighted.

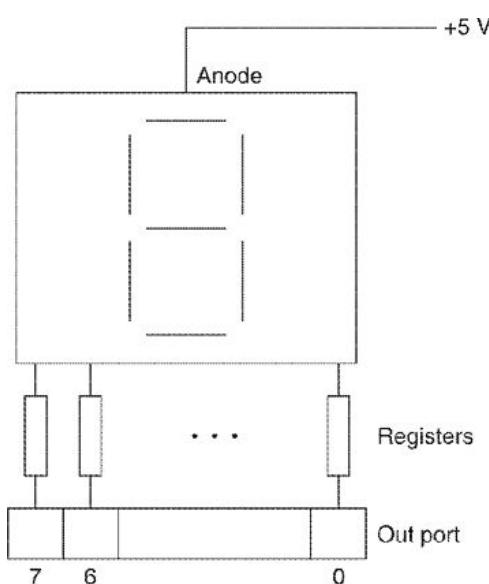


Figure 7.32 Microprocessor interface to seven-segment LED (parallel interface).

Serial interface

The parallel interface, as described above, is fast, requires minimum extra hardware for interfacing and the software is also very simple. However, this is not always the case.

- A single seven-segment display can be used for only a single character display. However real-life situations call for a large number of seven-segment displays. This requires extra ports (number of ports = number of seven-segment displays) to be defined, i.e. extra hardware. As an example, an application requiring 64 digits to be displayed, will require 64 ports to be defined, thus increasing the hardware complexity.
- The fast speed of parallel interface is not really required since the human eye, due to its limitations, cannot perceive such fast changes. In fact, if the display continuously changes very fast, the eye will not be able to read anything.

The serial interface which overcomes the disadvantages of the complex hardware in case of a large number of seven-segment displays, is shown in Figure 7.33.

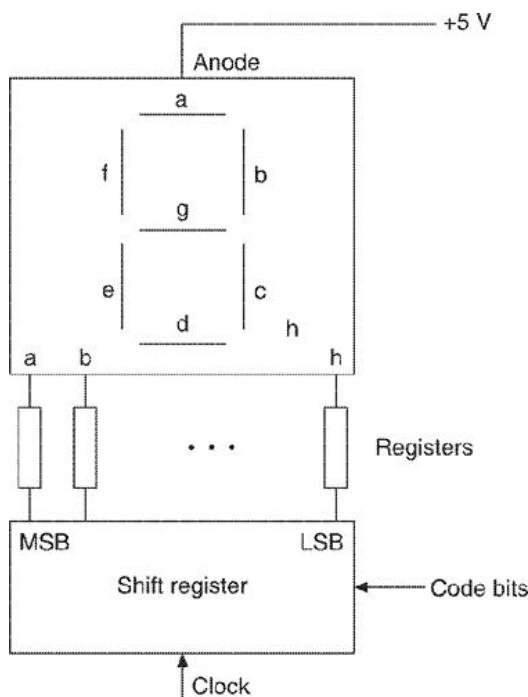


Figure 7.33 Microprocessor interface to seven-segment LED (serial interface).

The flowchart in Figure 7.34 shows the operation sequence which displays a digit in a seven-segment LED.

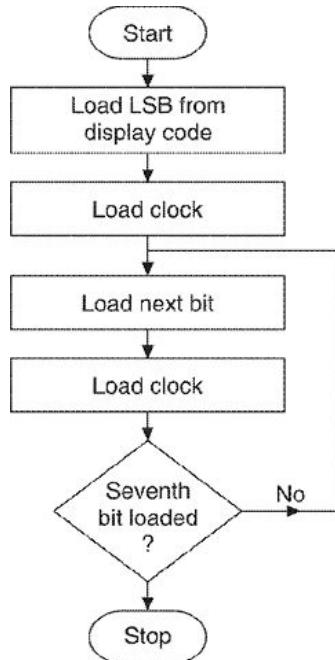


Figure 7.34 Serial interface of seven-segment LED to microprocessor—software flowchart.

As an example, let us assume that we want to display a character A. For character A, the segments a, b, c, e, f and g should glow. Thus, the display code for A will be

7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0
h	g	f	e	d	c	b	a

The zero bits in the display code will light the corresponding segments.

At the first step, LSB, i.e. the code for segment ‘a’ will be loaded in shift register. When the clock is loaded, the bit is passed to LSB of the shift register. When the second bit of the code (corresponding to segment ‘b’) is loaded in the shift register with the clock, the earlier bit shifts one step left and the later bit is included as the LSB of the shift register. The shifting occurs each time a bit is loaded. When the seventh bit has been loaded, the LSB of the display code reaches the MSB of the shift register and the character A appears. The character starts appearing as soon as the first bit is loaded in the shift register, but due to the fast execution and the limitation of the eye, the final character seems to appear instantly.

Serial interface of the seven-segment LED display requires only two output lines—one for the clock and the other for the code bits. This can be achieved through a single output port. Interestingly, the increase in the

number of the seven-segment displays has no effect on the number of lines. The shift registers are connected serially (Figure 7.35). Considering the limitations of the eye, it will still look instantaneous, except that the number of bits that will be entered are eight times the number of seven-segment LED displays interfaced. To display a character in display D₀ in Figure 7.35, 32 clock pulses will be required. However, the users will soon realize that a large number of displays can be easily interfaced, without any burden on the speed, by the serial interface technique.

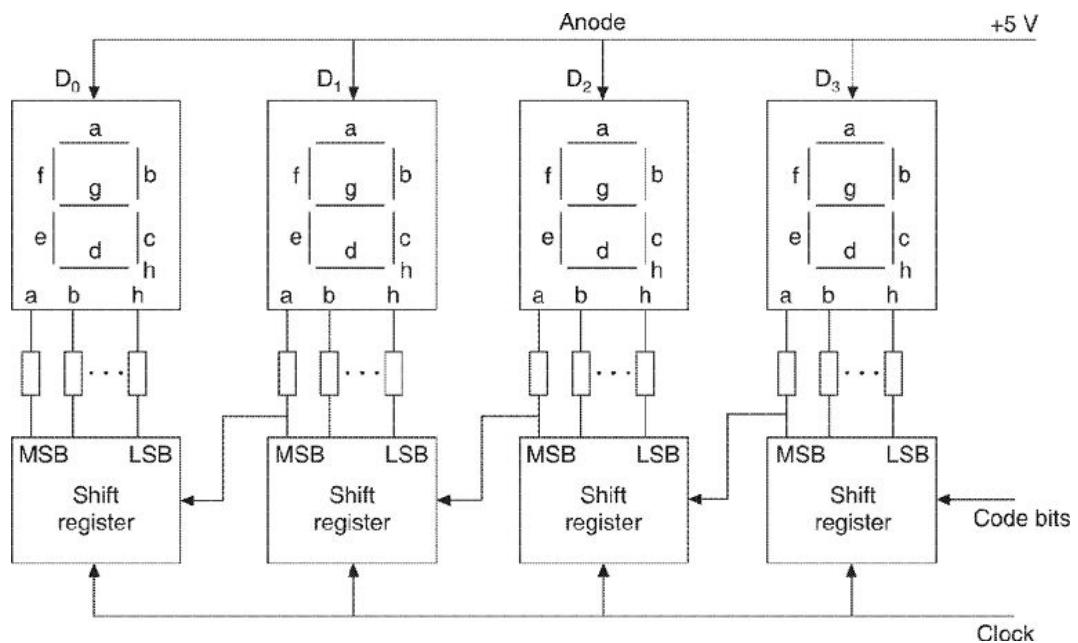


Figure 7.35 Cascading of seven-segment LEDs in a serial microprocessor interface.

The keyboard and the display are very important parts of any system. A system may have a few push-button keys or a large keyboard. Similarly, simple LEDs on the front panel as indicators to large displays containing seven-segment LEDs are also possible. We shall now briefly discuss how the keyboards and the seven-segment LEDs may be interfaced to the 8085 and the 8086 microprocessors. It is worth mentioning at this point that since the keyboard and the display control will be executed by the microprocessor software, such systems will be simple.

7.5.4 The 8085 Interfacing to Keyboard and Display

Figure 7.36 shows the interface of an 8 × 8 key keyboard to the 8085 through the 8255. The two ports A and B have been used for columns and rows, similar to Figure 7.23. The codes for

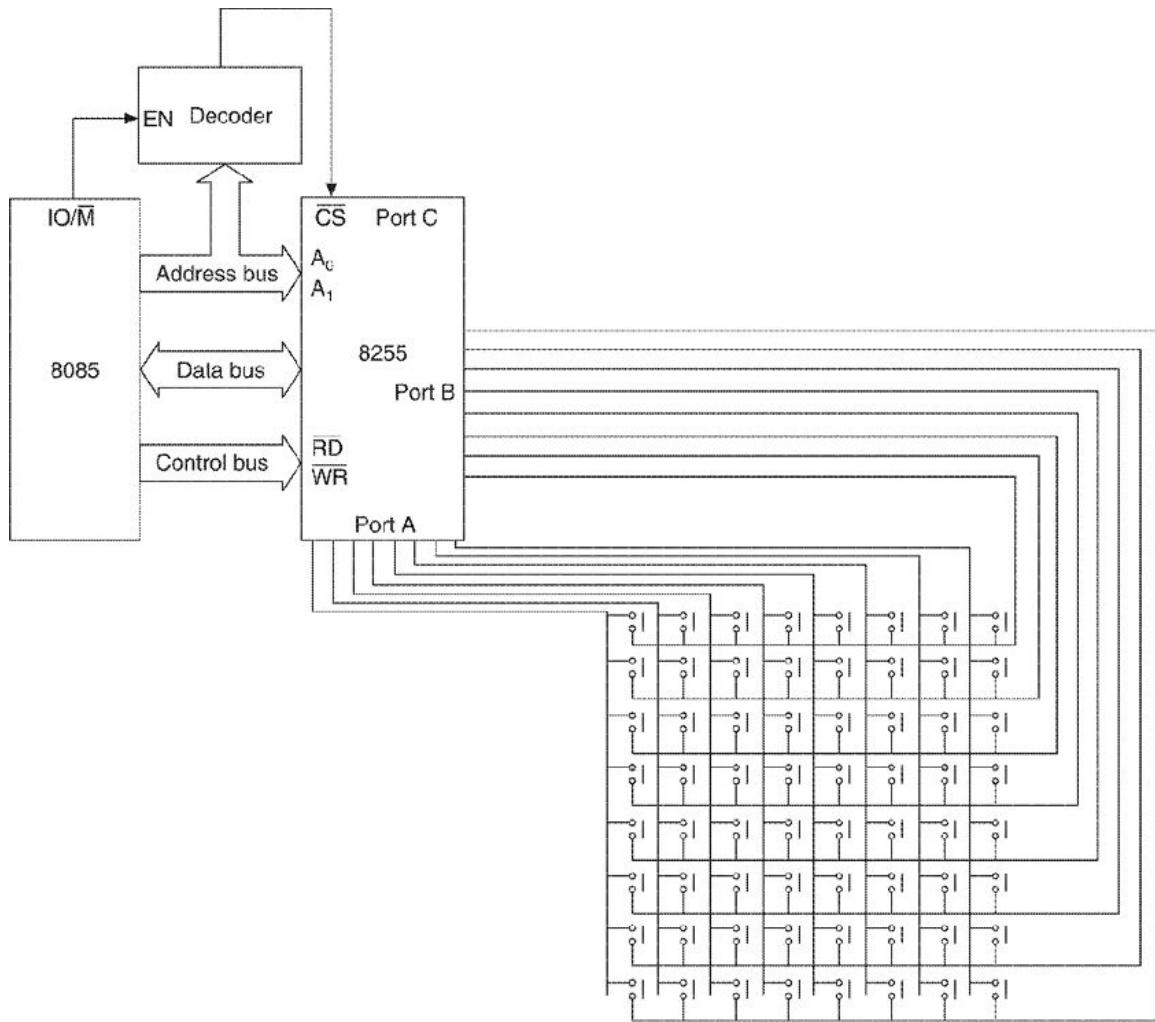


Figure 7.36 Keyboard interfacing to the 8085 using the 8255 PPI.

different keys are stored in the 8085 memory as look-up table. The microprocessor will read the corresponding code from memory and initiate action as per the requirement of the application.

Figure 7.37 shows the interface of four seven-segment displays, cascaded in serial mode, with the 8085 using the 8255. Only two lines of the 8255 are used as clock and data.

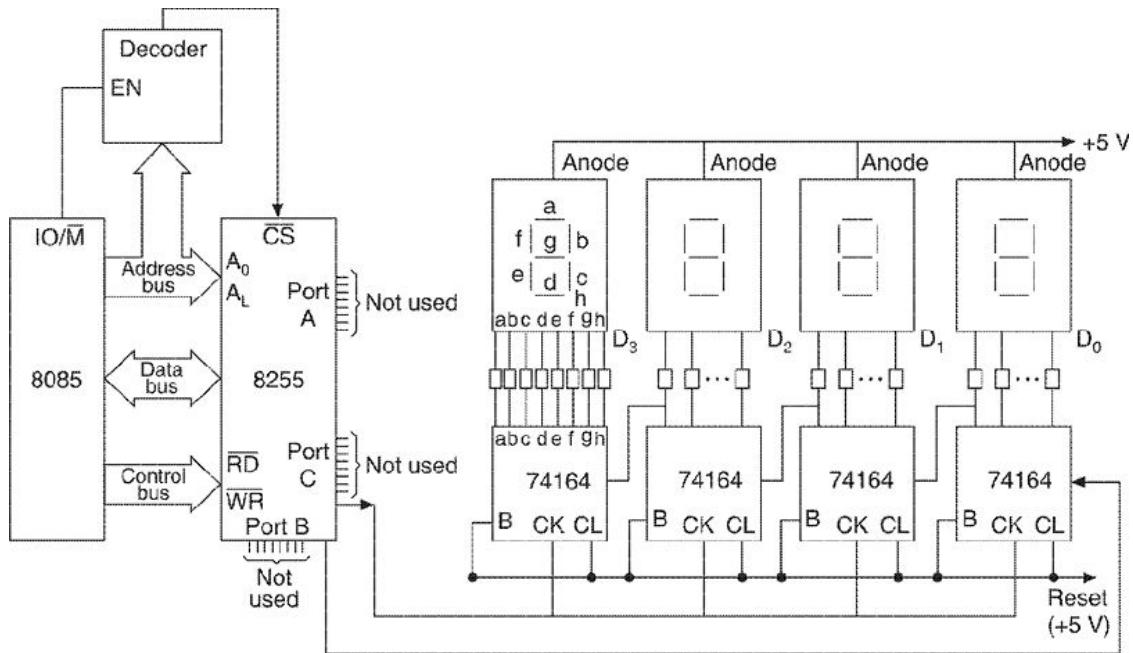


Figure 7.37 Seven-segment LED display interface to the 8085 using the 8255 PPI.

7.5.5 The 8086 Interfacing to Keyboard and Display

The interfacing of the 8086 with an 8 \times 8 key keyboard is shown in Figure 7.38. The 8086 is being used in minimum mode. Port A of the 8255 PPI is being used for the columns, whereas Port B is being used for the rows. The 8086 will determine the code of the depressed key using the look-up table stored in its memory and only then it will initiate the action.

The interfacing of the seven-segment LED display (4 in number) in serial mode cascaded manner is shown in Figure 7.39. Only two port lines of the 8255 have been used for clock and data. Other lines are free for interfacing with other circuits of the system.

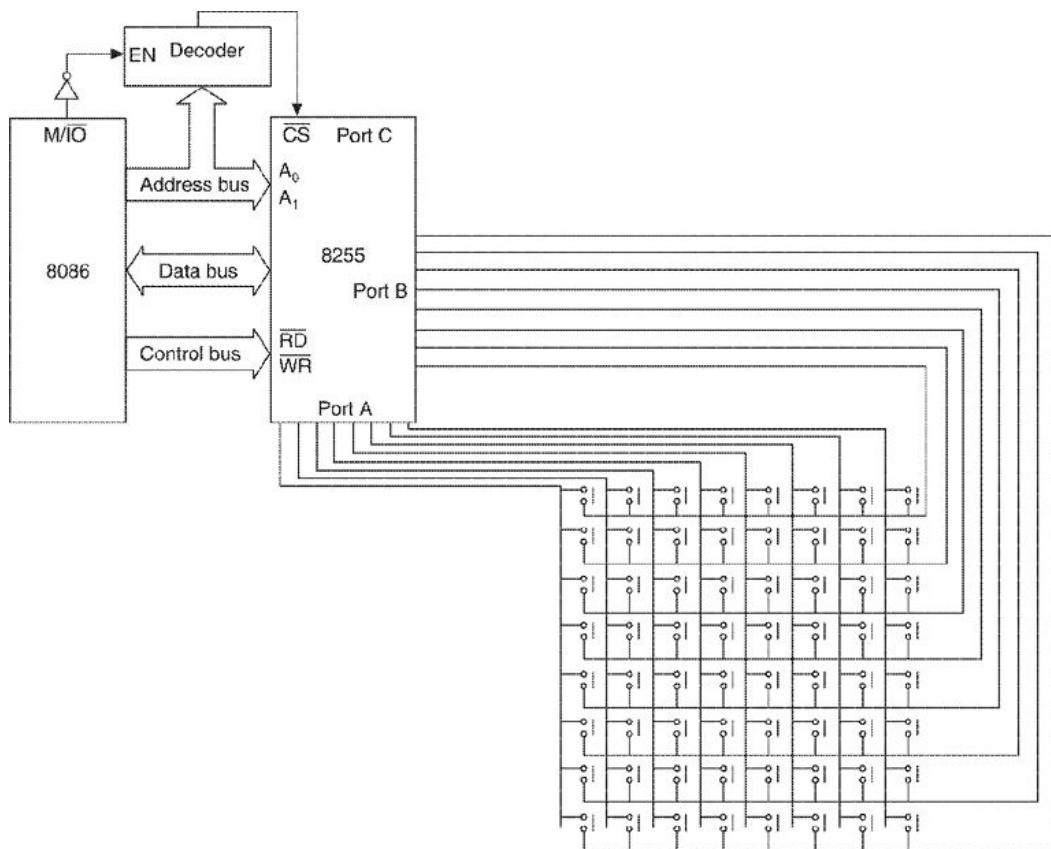


Figure 7.38 Keyboard interfacing to the 8086 using the 8255 PPI.

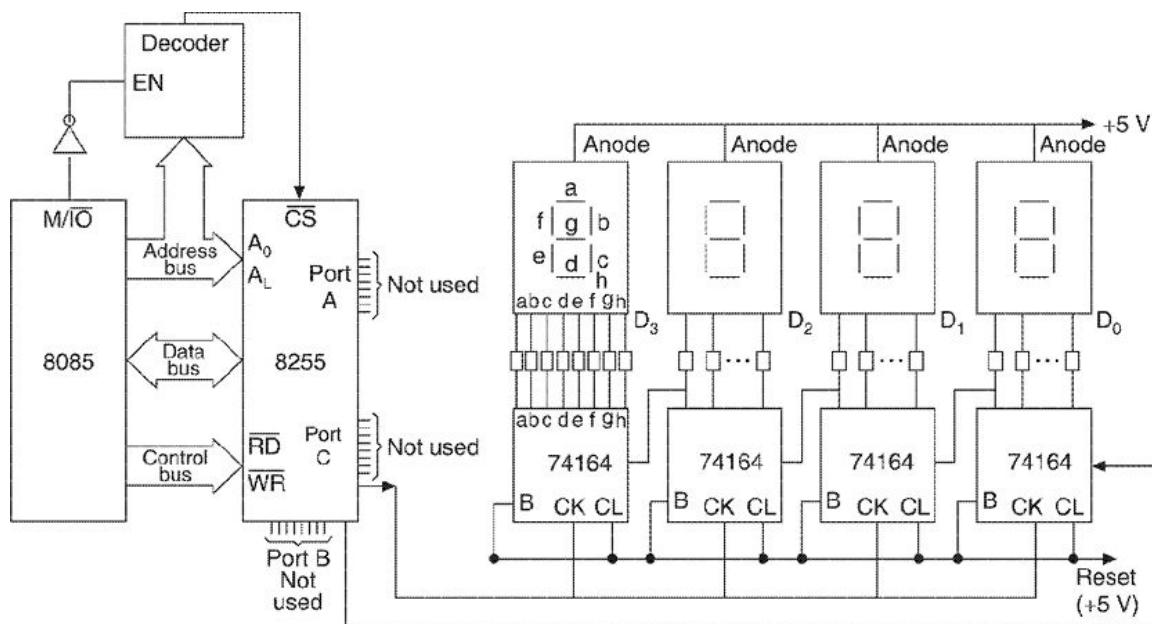


Figure 7.39 Seven-segment LED display interface to the 8086 using the 8255 PPI.

The flowchart for the keyboard and the display interfaces is shown in Figure 7.40 and Figure 7.41 respectively. The software is resident in the microprocessor memory and performs the task of scanning, debouncing

and finding the code for the key in case of keyboard interface, and sending various data bits and clock in case of display interface.

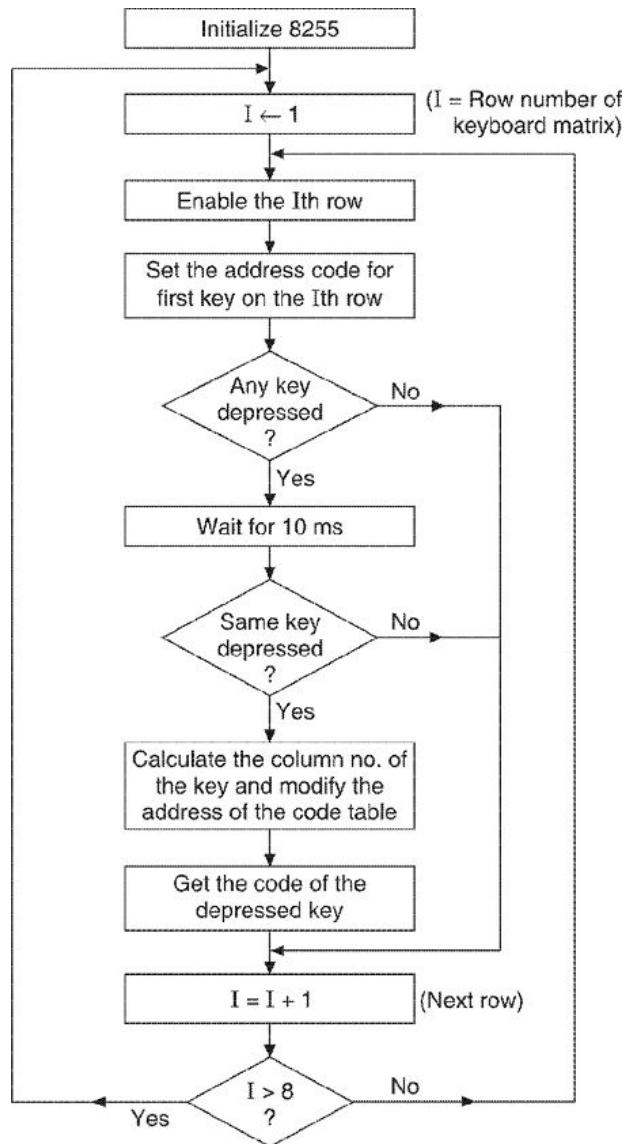


Figure 7.40 Flowchart for the 8 x 8 keyboard interface.

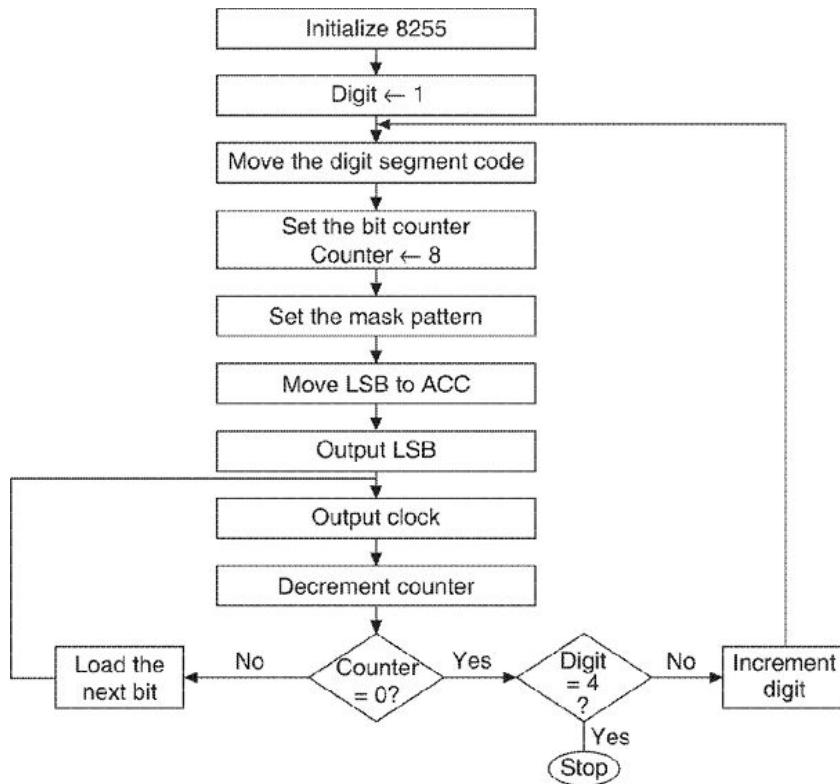


Figure 7.41 Flowchart for the seven-segment LED interface.

We shall now learn how to develop the software for these interfaces using the 8085 and 8086 microprocessors.

Using the 8085

SOFTWARE FOR KEYBOARD INTERFACE

```

;
; Intel 8255 Control Word = 1 0 0 1 0 0 0 0 = 90H, Port A = Input,
Port B = Output,
; Mode = 0
; Port Nos.—Port A = 00, Port B = 01, Port C = 02, Control Register
Port = 03
;
        MVI      A, 90H
        OUT     03 ; Initialize Intel 8255
LOOP0:    MVI      C, 01 ; Row Counter
        MOV      A, C
        STA     ZZZZ ; Store Row No. in ZZZZ
;
; Set address XXXX of Code Table (First Row).
;
        LOOP1:   LXI      H, XXXX

```

```

SUI      01
RAL
RAL
RAL      ; (A) = (Row No. - 1) □ 8
ADD      L
MOV      L, A    ; Address of first Row set
MOV      A, C
OUT     01
IN      00
ANA      A
JZ       LOOP2
MOV      B, A

;

; Delay loop.

;

DD:      LXI      D, YYYY
        DCX      D
        JNZ      DD
        IN       00
        CMP      B
        JNZ      LOOP2
        MVI      D, 00

        STC
        CMC      ; Clear Carry flag

RR:      RRC

        JC       CODE
        INR      D
        JMP      RR    ; Calculate Column No.

CODE:   MOV      A, D
        ADD      L
        MOV      L, A    ; Modify the address of Code Table
        MOV      A, M
        CALL     USE    ; Use the code for processing

;

; Increment the Row No.

;

LOOP2:  LDA      ZZZZ
        INR      A
        CPI      09

```

JZ	LOOP0
STA	ZZZZ
MOV	C, A
JMP	LOOP1
—	
—	
—	
ZZZZ	DB 0 ; Row No.
XXXX	DB —, —, —, —, —, —, —, — ; XXXX is Code
Table	

SOFTWARE FOR SEVEN-SEGMENT LED DISPLAY INTERFACE

```

;
; Intel 8085 software for seven-segment display
;
; Intel 8255 initialization
;
; Ports A, B, C are output ports.
; Control Word = 1 0 0 0 0 0 0 0 = 80H
; Port Nos.—A = 00, B = 01, C = 02, Control Register = 03.
; Circuit connections
; C0 (0th bit of port C) connected to clock, B0 (0th bit of port B)
connected to code input
;

        MVI    A, 80H
OUT    03
LXI    H, CODE ; CODE contains display codes in memory
MVI    D, 00 ; Digit No.
; LOOP1      MOV    A, D
        ADD    L
        MOV    L, A
        MOV    C, M ; Segment code in register C
        MVI    B, 08 ; Bit Counter
        MOV    A, C
        ANI    01
        OUT    01
;

; Output Clock.
;
; CLOCK:    MVI    A, 00
        OUT    02

```

```

        MVI      A, 01
        OUT     02
        MVI      A, 00
        OUT     02
;
; Repeat for the next segment.
;
        DCR      B
        JZ       MSB
        MOV      A, C
        RRC
        MOV      C, A
        ANI      01
        OUT     01H, AL
        JMP      CLOCK
;
; Repeat for the next digit.
;
MSB:      MOV      A, D
        CPI      03
        JZ       FINISH
        INR      D
        JMP      LOOP1
FINISH:    HLT
CODE       DB -, -, -, -

```

Using the 8086

SOFTWARE FOR KEYBOARD INTERFACE

```

;
; Intel 8255 Control Word = 1 0 0 1 0 0 0 0 = 90H , Port A = Input,
Port B = Output,
; Mode = 0
; Port Nos—Port A = 00, Port B = 01, Port C = 02, Control Register
Port = 03
;
        DATA SEGMENT
        ROW      DW    0
        COL      DB    0
        XXXX    DB    -, -, -, -, -, -, -, ; XXXX is
Code Table

```

```

        DATA    ENDS

        CODE   SEGMENT
                ASSUME CS: CODE, DS: DATA
                MOV AX, DATA
                MOV DS, AX
;
; Initialize Intel 8255.
;
                MOV AL, 90H
                OUT 03H, AL

        LOOP1:           MOV CX, 0001H ; Row Counter
        LOOP2: MOV ROW, CX
;
; Set address XXXX of Code Table (First Row) in BX.
;
                LEA BX, XXXX
                DEC CX ; (CX) = ROW No. - 1
                SAL CX, 1
                SAL CX, 1
                SAL CX, 1 ; (X) = (ROW No. - 1) □ 8
                ADD BX, CX ; (BX) = Address of first element of row
;
; Scan for key depression.
;
                MOV CX, ROW
                MOV AL, CL
                OUT 01H, AL
                IN AL, 00H
                AND AL, FFH
                JZ LOOP3 ; No key depressed. Scan the next row
;
; Key depressed.
;
                MOV DL, AL
;
; Delay the loop for debouncing.
;
                MOV CX, FFFFH
        DD:     NOP
        LOOP DD

```

```

;
; Again scan for key depression.
;
        IN    AL, 00H
        CMP   AL, DL
        JNZ   LOOP3 ; False key depression
;
; Key depression validated. Calculate the Column No.
;
        MOV   CL, 00
        CLC ; Clear carry flag
RR:      RCR   AL, 1
        JC    CODEIS
        INC   CL
        JMP   RR
;
; Find the key code.
;
CODEIS:   ADD   BX, CX ; (BX) = Address of key code
        MOV   AL, (BX)
        CALL  USE  ; Use the code for processing
        JMP   ENDIS
;
; Increment the Row No. and repeat.
;
LOOP3:    MOV   CX, ROW
        INC   CX
        CMP   CX, 0009H
        JZ    LOOP1
        JMP   LOOP2
ENDIS:   NOP
        CODE  ENDS

```

SOFTWARE FOR SEVEN-SEGMENT LED DISPLAY INTERFACE

```

;
; Intel 8086 software for seven-segment display
;
; Intel 8255 initialization
;
; Ports A, B, C are output ports.

```

```

; Control Word = 1 0 0 0 0 0 0 0 = 80H
; Port Nos.—A = 00, B = 01, C = 02, Control Register = 03.
; Circuit connections
; C0 (0th bit of port C) connected to clock, B0 (0th bit of port B)
connected to code input
;

        DATA SEGMENT
        DISP-CODE DB -, -, -, -, -, -, -; Display codes
        DIGIT     DB 0
        DATA ENDS

        CODE SEGMENT
        ASSUME CS: CODE, DS: DATA
        MOV AX, DATA
        MOV DS, AX

;
; Initialize Intel 8255.
;
        MOV AL, 80H
        OUT 03H, AL
        LEA BX, DISP-CODE ; (BX) = Address of DISP-
CODE
        MOV DX, 00 ; Digit No.

LOOP1:      ADD BX, DX ; (BX) = Address of digit code
to be displayed
        MOV AL, (BX)
        MOV DIGIT, AL
;

;
; Output the digit code bit-by-bit with clock.
;
        MOV CL, 08H
LOOP2:      AND AL, 01H
        OUT 01H, AL
;

;
; Output Clock.
;
        MOV AL, 00
        OUT 02H, AL
        MOV AL, 01
        OUT 02H, AL

```

```

        MOV  AL, 00
        OUT  02H, AL
;
; Repeat for the next segment.
;
        DEC  CL
        JZ   MSB
        MOV  AL, DIGIT
        RCR  AL, 1
        MOV  DIGIT, AL
        JMP  LOOP2
;
; Repeat for the next digit.
;
MSB:      CMP   DX, 0003H
        JZ    FINISH
        INC   DX
        JMP   LOOP1
FINISH:   NOP
CODE     ENDS

```

7.6 KEYBOARD AND DISPLAY CONTROLLER (INTEL 8279)

In the keyboard interface, the microprocessor scans the various keys, performs software debouncing and finally finds out the code of the depressed key. In the display interface, the microprocessor is busy in sending bit-by-bit information to the shift registers regarding the display of various segments. This puts load on the CPU, which it may not be able to take in certain environments where the CPU has to handle a number of tasks in a very short duration.

The system designer, therefore, needs an interface that can control these functions without placing the load on the CPU. The 8279 provides these functions for the microprocessors.

The 8279 (Figure 7.42) consists of two sections—keyboard and display. The keyboard section can interface to regular typewriter style keyboards or random toggle or thumb switches. The display section drives the alphanumeric displays or a bank of indicator lights. Thus, the processor is relieved from scanning the keyboard or refreshing the display.

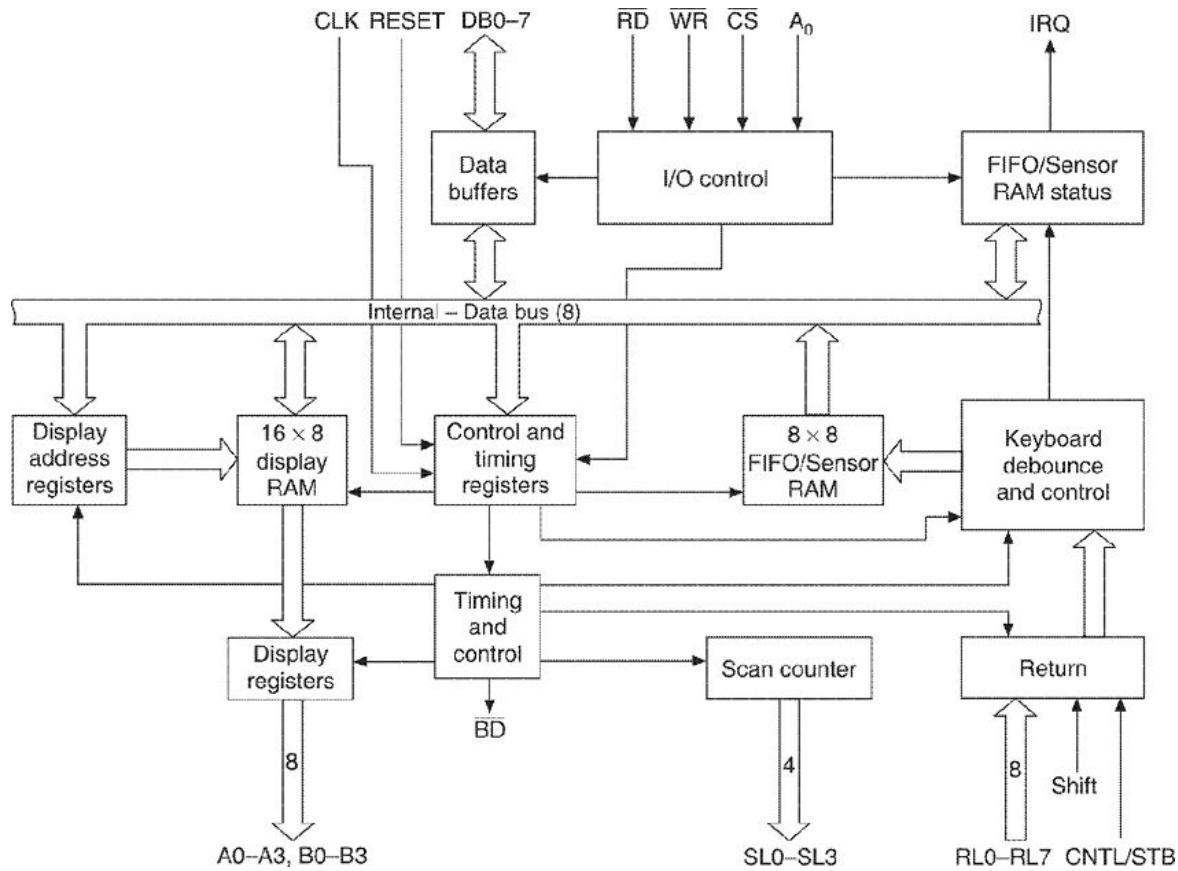


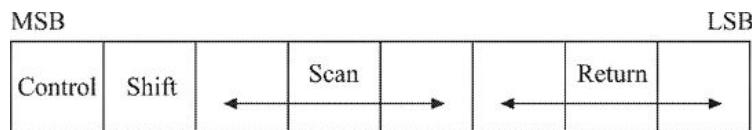
Figure 7.42 Internal block diagram of the 8279 chip.

7.6.1 Keyboard Section

The 8279 offers four scan lines **SL0–SL3**, eight return lines **RL0–RL7**, a shift line and a control line. The scan lines are interfaced to the rows of the keyboard. The return lines can be directly connected to the eight columns of the keyboard, whereas the shift and control lines are connected to the shift and control keys. When a keyboard is used along with shift and control keys, four states may be defined by each key in the keyboard, namely when the control and shift keys are not depressed, when the control key is depressed, when the shift key only is depressed, and when both of them are depressed. Thus, four characters/functions may be defined by each key.

There are two ways in which scan lines can be interfaced to the keyboard—encoded or decoded scan. With the encoded scan, the row number is output on the three low-order scan lines and a 3:8 decoder is needed to send signals to the individual rows. This enables the interfacing of 8×8 keyboard as shown in Figure 7.43. With the decoded scan, the 8279 provides row activation signals on four lines in sequence which can be directly interfaced to four rows of keyboard thus enabling the interfacing of 4×8 keyboard (Figure 7.44). In both the cases a key

depression generates a 6-bit encoding of the key position, a shift bit and a control bit. The data format is shown below.



This information is entered in the RAM resident in the 8279. The RAM serves as a simple queue, i.e. First In First Out (FIFO), meaning that the position code for various key depressions will be stored and transferred to the computer in order of their entry. When FIFO is empty and a key depression occurs, when the interrupt signal is sent to the processor on the IRQ line.

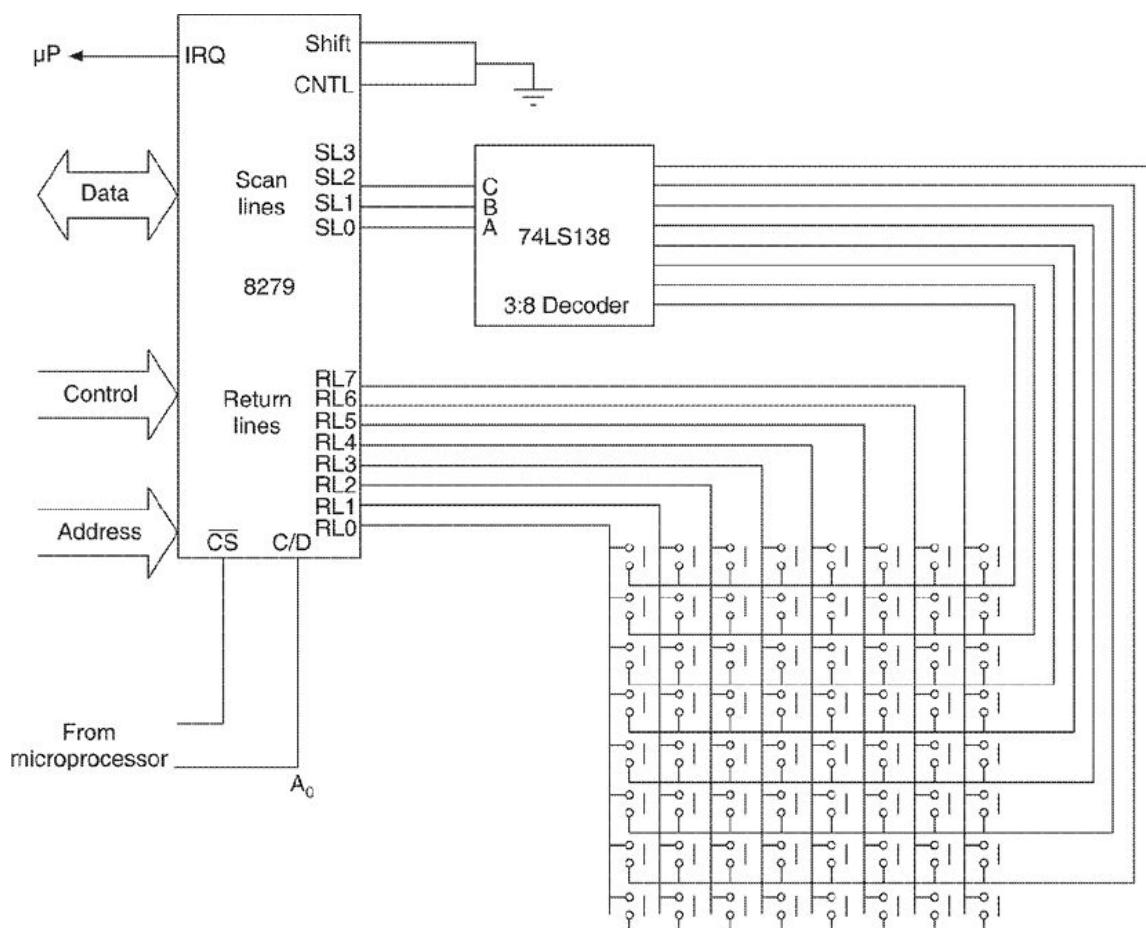


Figure 7.43 Interfacing 8 x 8 keyboard to the 8279 chip: encoded scan.

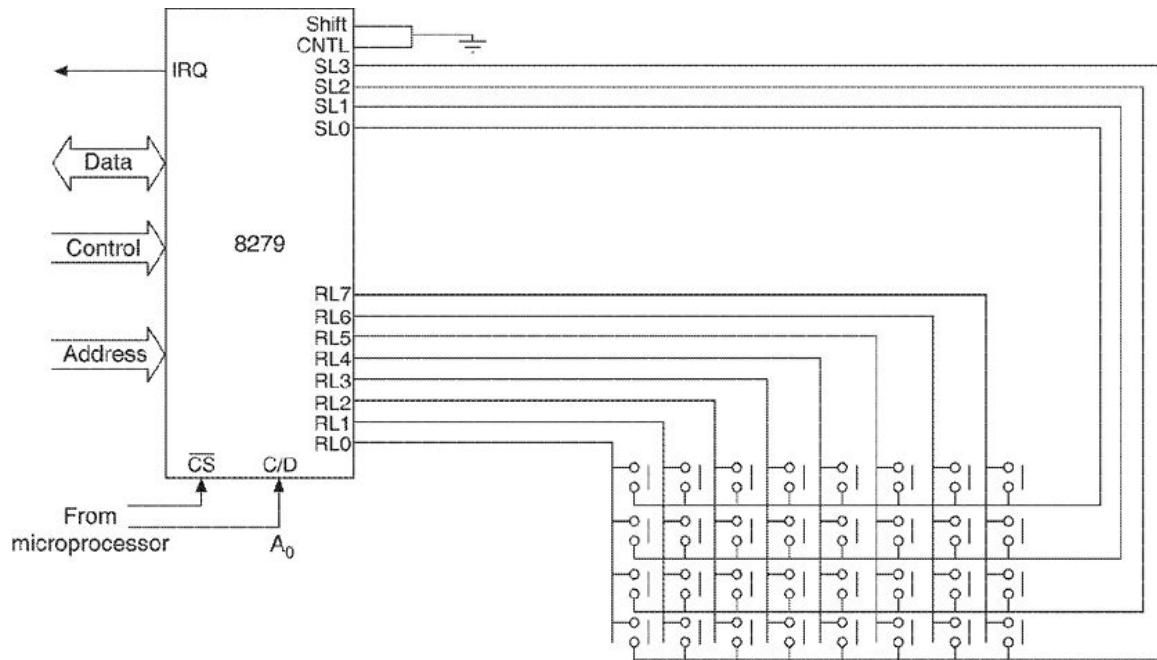


Figure 7.44 Interfacing 8 × 4 keyboard to the 8279 chip: decoded scan.

Every key depression should be validated before entering the information in FIFO. The 8279 has two types of debounce: 2-key lockout and N-key rollover. In case of 2-key lockout, the debounce logic is set as soon as a key is depressed. A full scan of the keyboard is ignored (i.e. the 8279 waits), then other depressed keys are looked for.

If no other depression is encountered, it is taken as a single key depression and the information is entered in FIFO. If two or more keys are pressed simultaneously, the key which is released last, is treated as a single depression. In case of N-key rollover, each key depression is treated independently from all the others, i.e. more than one key can be pressed simultaneously and recognized. The debounce logic checks for only the false depression by waiting for the 2-keyboard scan period and then checking the same key to find out whether it is still depressed. If it is, the information is then entered in FIFO. Any number of keys can be depressed and entered in FIFO.

So far, we have discussed the keyboard where depending on the position of the depressed key, the processor initiates some action. However, the same scan and return lines can be used to interface a matrix of switches. These switches have been named sensors and have only two states—On and Off. With the encoded scan the 8 × 8 sensor matrix, and with the decoded scan the 4 × 8 sensor matrix (Figure 7.45), can be interfaced (shift and control lines are not used in this case). The return lines contain the data regarding the switch positions (Off or On) in the scanned row of the sensor matrix. The data format is shown below.

MSB	RL7	RL6	RL5	RL4	RL3	RL2	RL1	RL0	LSB
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

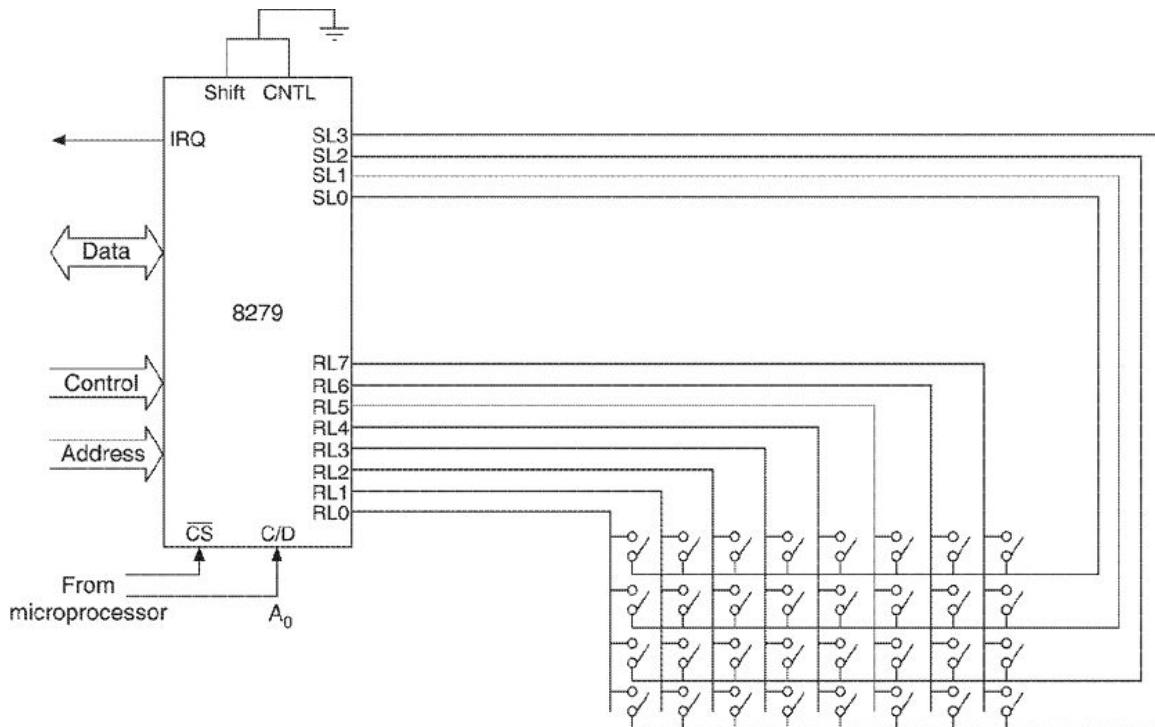


Figure 7.45 Interfacing the 8x4 sensor matrix to the 8279 chip: decoded scan.

This information is entered directly into the sensor RAM. In this way, the sensor RAM keeps on the image of the state of the switches in the sensor matrix. If a change in the sensor value is detected, the interrupt line (IRQ) goes high. There is no debouncing in this mode.

Apart from the keyboard and the sensor matrix interfacing, these lines can also be used for general purpose strobed input. This data is also entered to FIFO from the return lines. The data entry is caused by the rising edge of the CNTL/STB line pulse. The data format is as follows:

RL7	RL6	RL5	RL4	RL3	RL2	RL1	RL0
-----	-----	-----	-----	-----	-----	-----	-----

Data can also come from the switch matrix or another encoded keyboard.

Thus, the following modes are possible with the keyboard section.

- Encoded Scan Keyboard—2-key lockout
- Encoded Scan Keyboard—N-key rollover
- Decoded Scan Keyboard—2-key lockout
- Decoded Scan Keyboard—N-key rollover

- Encoded Scan Sensor Matrix
- Decoded Scan Sensor Matrix
- Strobed Input

7.6.2 Display Section

The display section provides a scanned display interface for LED, incandescent and other popular display technologies. It allows both numeric and alphanumeric segment displays as well as lamp indicators. The 8279 uses four scan lines SL0–SL3, two 4-bit (A0–A3 and B0–B3) output ports, a 16×8 display RAM and a blank display (\overline{BD}) line for its display operation. The two 4-bit ports can be used independently or as one 8-bit port. For BCD type seven-segment LED displays, only 4-bit codes are needed in the BCD form. The two 4-bit ports can be used independently for this.

The 16×8 display RAM can be organized into a dual 16×4 RAM. Thus the 8279 facilitates 8 or 16 characters multiplexed displays that can be organized as dual 4 bit or single 8 bit. Both right and left entry display formats are possible.

The display RAM can be loaded or read by the CPU. The read/write addresses are programmed by the CPU command. They can also set auto-increment after each read or write. The display RAM can be directly read by the CPU after the correct mode and address is set. The 8279 interface to display is shown in Figure 7.46. It should be noted that when the keyboard is in the decoded scan, the display will automatically be in the decoded scan, i.e. only the first 4 characters in the display RAM will be displayed. The \overline{BD} output is used to blank the display during digit switching. The display-blanking command can be used to activate the \overline{BD} line, i.e. to blank the display.

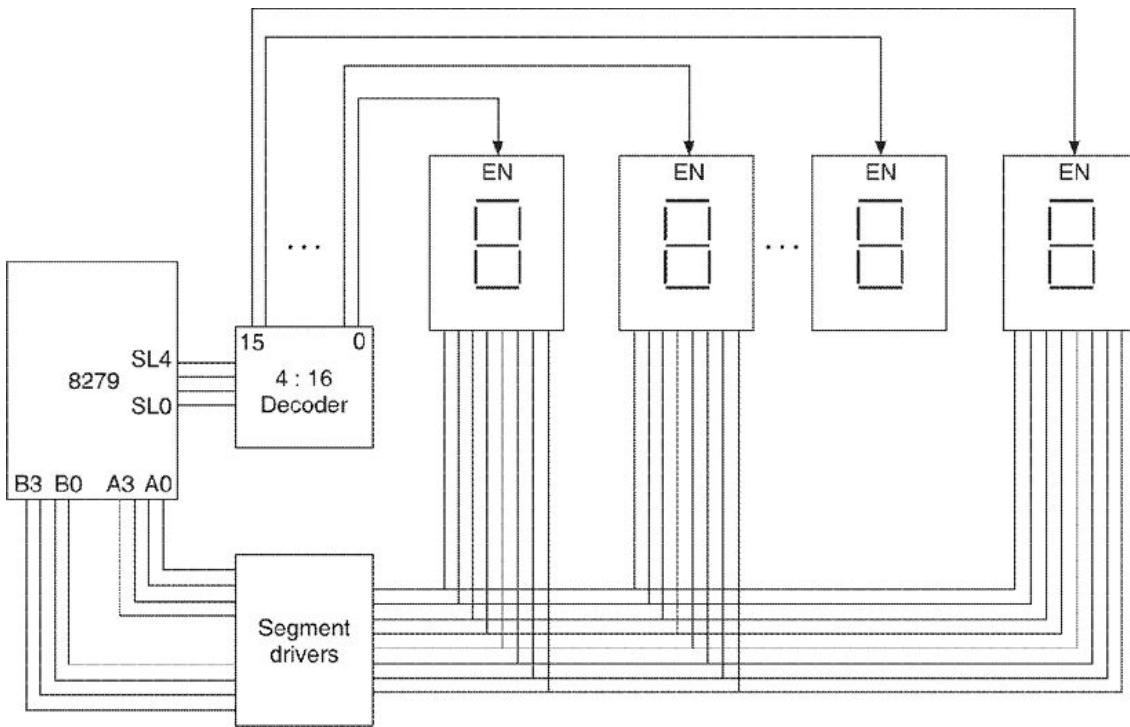


Figure 7.46 Interfacing the seven-segment LED display to the 8279 chip.

7.6.3 Software Commands

Following commands are used to program the 8279 operating modes to perform read/write, to perform display blanking, etc. The 8279 differentiates between the data and the command by sensing the A₀ input line. If A₀ is high, the input byte is taken as command, otherwise as data.

Keyboard/display mode

MSB	0	0	0	D	D	K	K	K	LSB
DD									

DD

- 00—8-8-bit character display—Left entry
- 01—16-8-bit character display—Left entry
- 10—8-8-bit character display—Right entry
- 11—16-8-bit character display—Right entry

KKK

- | | |
|------------------|-------------------------|
| 000—Encoded Scan | Keyboard—2-key lockout |
| 001—Decoded Scan | Keyboard—2-key lockout |
| 010—Encoded Scan | Keyboard—N-key rollover |
| 011—Decoded Scan | Keyboard—N-key rollover |
| 100—Encoded Scan | Sensor Matrix |
| 101—Decoded Scan | Sensor Matrix |

110—Strobed Input	Encoded Display Scan
111—Strobed Input	Decoded Display Scan

When the decoded scan is set in the keyboard mode, the display is reduced to (decoded scan) 4 characters independent of the display mode set.

Program clock

MSB	LSB							
0	0	1	P	P	P	P	P	P

PPPPP is a scalar value set between 2 to 31. The external clock is divided by PPPPP internally to get the basic internal frequency. One must choose a value of PPPPP (depending on the external clock frequency of the processor), so as to obtain 100 kHz internal frequency, which gives the specified scan and debounce time.

Read FIFO/Sensor RAM

MSB	LSB							
0	1	0	AI	X	A	A	A	A

The AI is the auto-increment flag for sensor RAM only, and AAAA is the row number in FIFO/Sensor RAM which is being read. If AI is 1, the row select counter will be incremented after each read, so the next read will be from the next sensor row. This auto incrementing has no effect on the display.

Read display RAM

MSB	LSB							
0	1	1	AI	A	A	A	A	A

The AI is the auto-increment flag for display RAM, and AAAA is the address of the character to be read in display RAM. Since the CPU uses the same counter for both read and write, this command also sets the next write location and the auto-increment mode.

Write display RAM

MSB	LSB							
1	0	0	AI	A	A	A	A	A

The AI is the auto-increment flag, and AAAA is the address of the display RAM location where the write will be performed.

Display write inhibit/blanking

MSB							LSB
1	0	1	X	IW	IW	BL	BL

The IW is inhibit writing (nibble A or B) and BL is blanking (nibble A or B).

Clear

MSB							LSB
1	1	0	CD2	CD1	CD0	CF	CA

The CD is clear display, CF is clear FIFO status and CA is clear all. The CD is used to clear all the positions of the display RAM by the programmable code as follows:

CD1	CD0	Action
0	X	—All zeros (X = Don't Care)
1	0	—20H
1	1	—All ones

CD2 enables clear display when = 1. The display can also be cleared by CA = 1.

End interrupt mode/error mode set

MSB							LSB
1	1	1	E	X	X	X	X

In the sensor matrix mode this command lowers the IRQ line and enables further writing into the RAM. For the N-key rollover mode, if E = 1, the 8279 will operate in the special error mode (*MCS'85 Users Manual*, Intel Corporation, 1983).

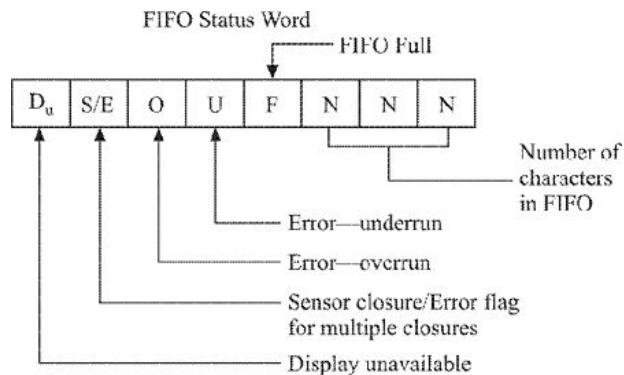
FIFO status

FIFO status is used in the Keyboard and Strobed input modes to indicate the number of characters in the FIFO and to indicate whether an error has occurred. There are two types of errors possible: overrun and underrun. Overrun occurs when the entry of another character into a full FIFO is attempted. Underrun occurs when the CPU tries to read an empty FIFO.

The FIFO status word also has a bit to indicate that the Display RAM was unavailable because a Clear Display or Clear All command had not completed its clearing operation.

In a Sensor Matrix mode, a bit is set in the FIFO status word to indicate that at least one sensor closure indication is contained in the Sensor RAM.

In Special Error Mode the S/E bit shows the error flag and serves as an indication to whether a simultaneous multiple closure error has occurred.



7.6.4 Interfacing the 8279

Figure 7.47 shows the interfacing of the 8085 with 8 × 8 keyboard and 16-character LED display using the 8279 keyboard and the display controller. In case of the 8086 interfacing to 8 × 8 keyboard and 16-character display using the 8279 keyboard display controller, IRQ may be connected to the INTR. Octal buffer 74ALS244 has been used to transfer interrupt type 20H to the data bus in response to $\overline{\text{INTA}}$ signal. Also, the $\text{M}/\overline{\text{IO}}$ signal will be inverted before connecting to the EN pin of the decoder (Figure 7.48).

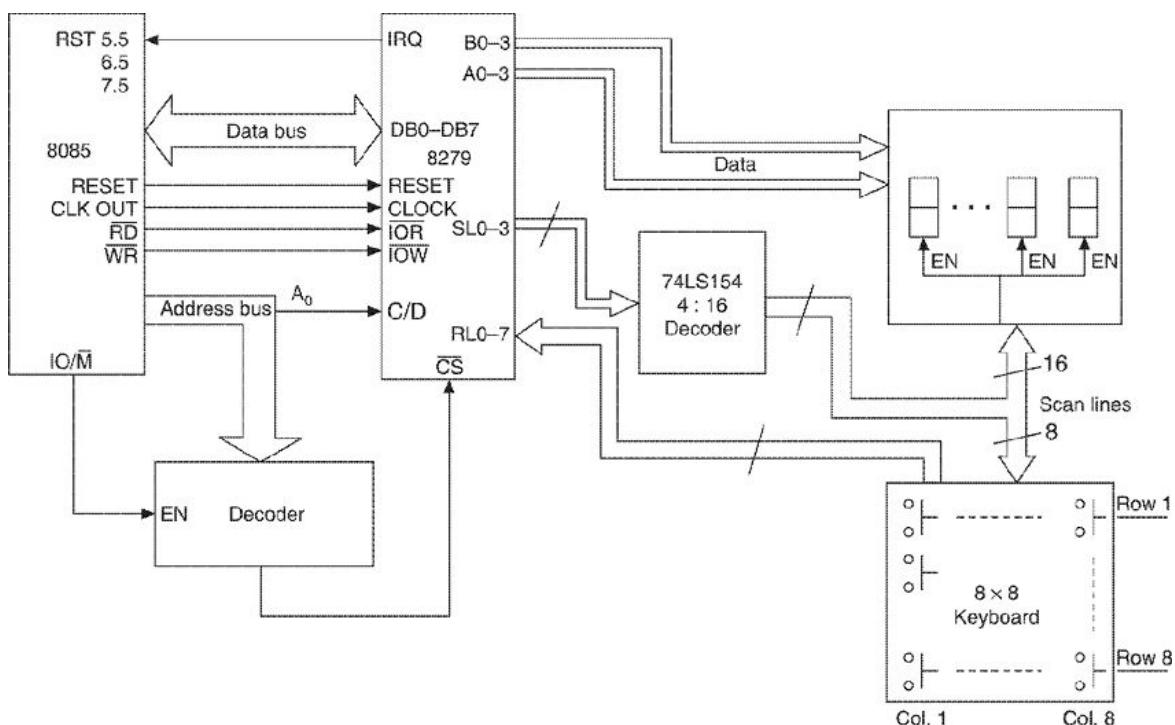


Figure 7.47 Keyboard and seven-segment display interface to the 8085 using the 8279.

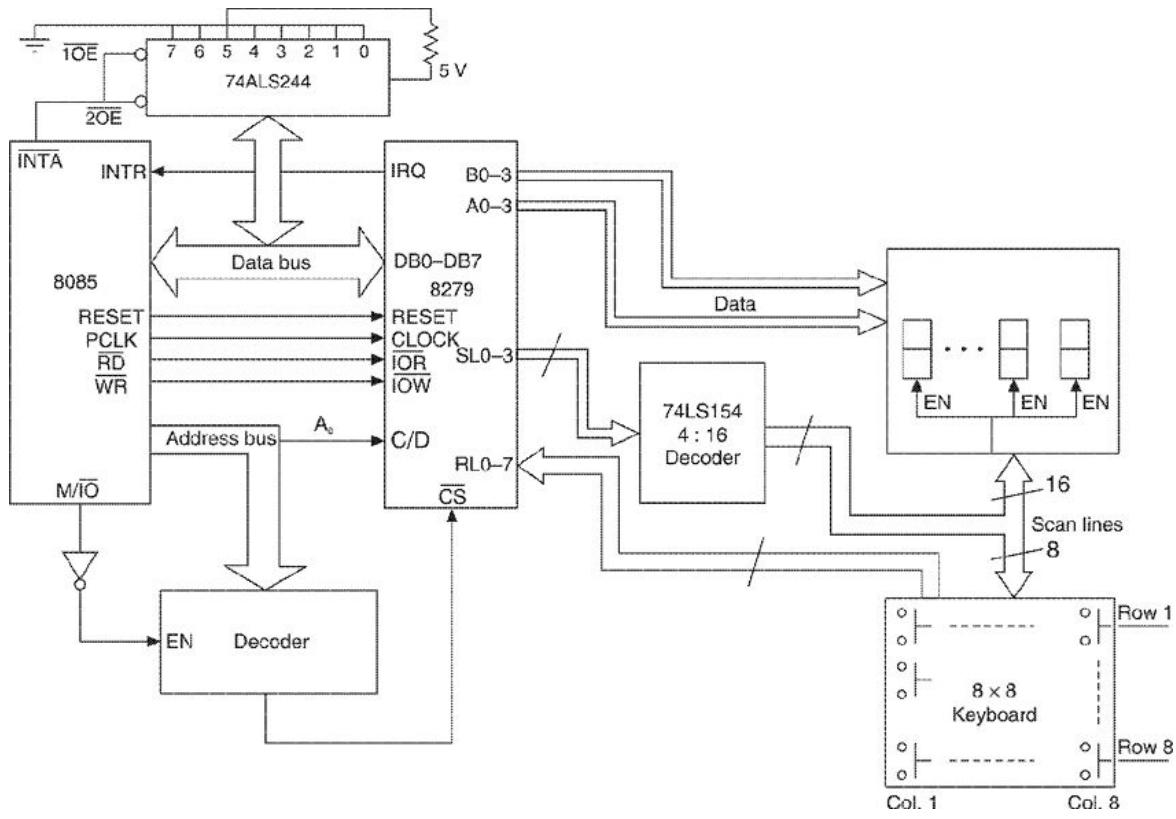


Figure 7.48 Keyboard and seven-segment display interface to the 8086 using the 8279.

The flowchart for the keyboard interface is shown in Figures 7.49(a) and (b). In the flowchart in Figure 7.49(a), the IRQ output is used to interrupt the processor and is connected to one of the interrupt pins (RST 5.5, 6.5, 7.5 in the 8085 or INTR in the 8086). In case of Figure 7.49(b), the IRQ output is not used.

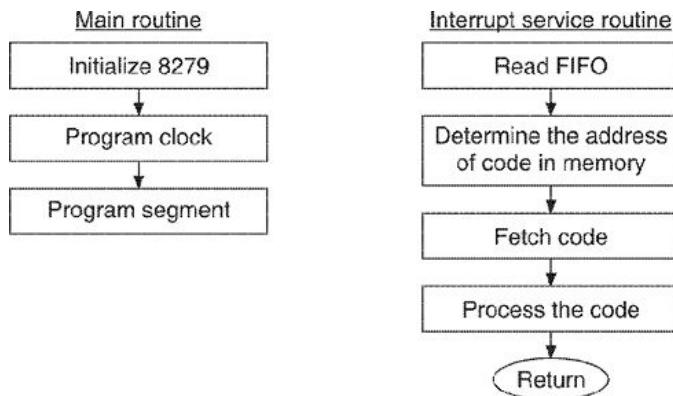


Figure 7.49(a) Flowchart for microprocessor interface to keyboard using the 8279 (IRQ used).

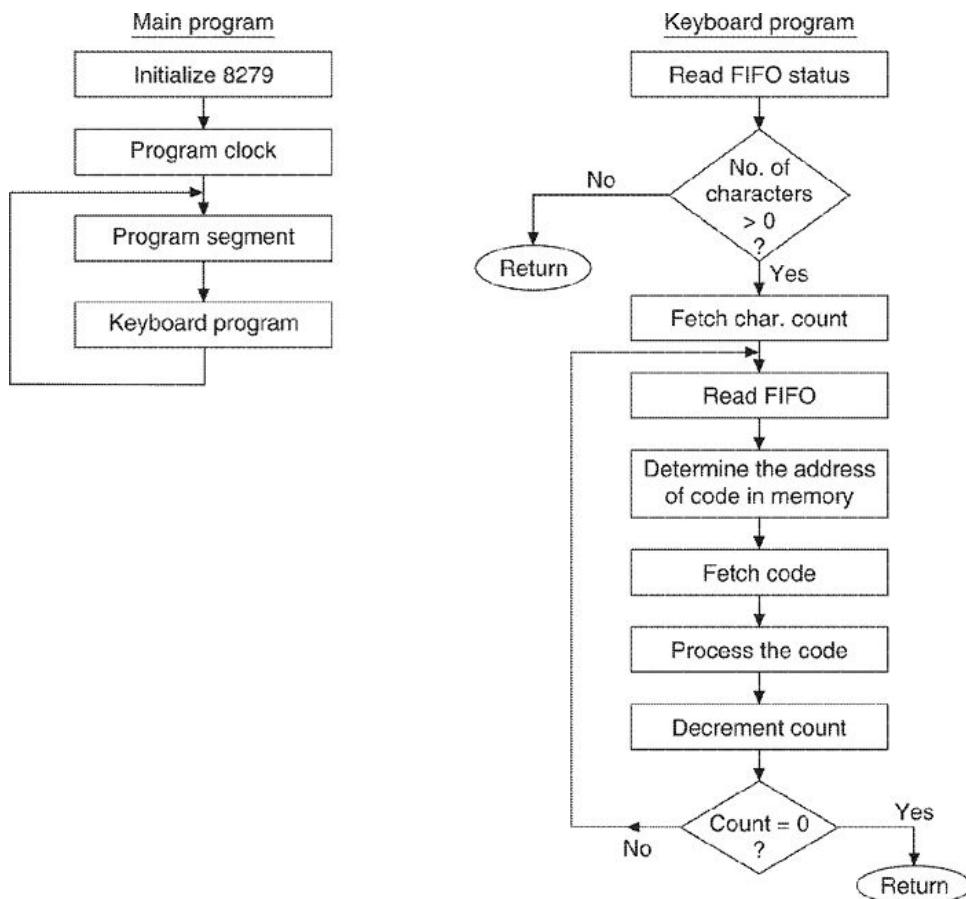


Figure 7.49(b) Flowchart for microprocessor interface to keyboard using the 8279 (IRQ not used).

This interface programs for the 8085 and 8086 microprocessors are given below.

The 8085 program for Figure 7.49(a)—IRQ used

Main Program

; For data read	$A_0 = 0$ and $\overline{RD} = \text{low}$;
; For FIFO status read	$A_0 = 0$ and $\overline{RD} = \text{low}$
; For command output to 8279	$A_0 = 1$ and $\overline{WR} = \text{low}$
; For data write to 8279	$A_0 = 0$ and $\overline{WR} = \text{low}$

X1	EQU —; Command port
X0	EQU —; Data input port
Y1	EQU —; Status input port
YY	DB —; Program clock command
Y0	EQU —; Data input port
MVI	A, 00H ; Mode set command
OUT	X1 ; Command output
LDA	YY ; Program clock command

OUT X1

— —

— — ; Program Segment

— —

Interrupt Service Routine

LXI H, ZZZZH

:

; ZZZZH = Starting address of Code Table in memory

;

LDA YY

OUT X1

;

; Read FIFO from 00 location in auto-increment mode (0 1 0 1 0 0 0 0
= 50H).

;

MVI A, 50H

OUT X1

IN X0

CALL FIND

CALL PROG

RET

Subroutine FIND

; Scanned keyboard data in register A is in the following format—

; D7—CNTL, D6—SHIFT, D5 D4 D3—SCAN, D2 D1 D0—

RETURN

;

FIND: MOV B, A ; Move scanned keyboard
data to register B.

ANI 07H ; (A) = Column No.

MOV C, A

MOV A, B

ANI 38H ; (A) = 8 □ Row No.

ADD L

ADD C ; Add Column No.

MOV L, A ; (H)(L) = Address of the depressed
key

MOV A, M ; (A) = Code of the depressed key

RET

Subroutine PROG

```
;  
; Subroutine contains program to process the code of the  
depressed  
; key  
;
```

The 8085 program for Figure 7.49(b)—IRQ not used

```
;  
; Intel 8085 software for the keyboard using the 8279 when IRQ is not  
used  
;  
; For data read A0 = 0 and RD = low  
; For FIFO status read A0 = 0 and RD = low  
; For command output to 8279 A0 = 1 and WR = low  
; For data write to 8279 A0 = 0 and WR = low  
;  
X1 EQU —; Command port  
X0 EQU —; Data input port  
Y1 EQU —; Status input port  
YY DB —; Program clock command  
Y0 EQU —; Data input port  
;  
; Initialize Intel 8279  
MVI A, 50H  
OUT X1  
;  
; Program Clock  
;  
LDA YY  
OUT X1  
LOOP: —  
—  
—  
CALL KBDPG  
—  
—  
—  
JMP LOOP  
;
```

```

; Keyboard Subroutine
;
KBDPG:           LXI    H, ZZZZ
                IN     Y1
;
; (A) = FIFO Status, D2 D1 D0 = No. of Characters
;
                ANI    07H
                RZ     ; Return if zero
                MOV    C, A ; (C) = Count
; Read FIFO from 00 location in auto-increment mode.
; Command = 01010000 = 50H
LOOP1       MVI    A, 50H
            OUT   XI
            IN    X0
            CALL  FIND
            CALL  PROG
            DCR   C
            JNZ   LOOP1
            RET

```

The 8086 program for Figure 7.49(a)—IRQ used

```

; Program for keyboard interfacing of the 8086 using the 8279
; IRQ output of the 8279 is connected to INTR pin of the 8086 to
generate Type 32,
; i.e. 20H interrupt.
; Interrupt vector table of the 8086 will have ISR addresses as
follows:
; Segment address at 0082H, 0083H
; Offset address (IP contents) at 0080H, 0081H
; ISR is named as procedure KBDP
; For data read          A0 = 0 and RD = low
; For FIFO status read    A0 = 0 and RD = low
; For command output to 8279 A0 = 1 and WR = low
; For data write to 8279  A0 = 0 and WR = low
; Main Program
;
DATA SEGMENT
X1      EQU —; Command port for Intel 8279

```

```

X0      EQU —; Data input port for Intel 8279
Y1      EQU —; Status input port
YY      DB  —; Command for programming clock
COL    DW   0
ROW    DW   0
TEMP   DB   0
ZZZZ   DB   —, —, —, —
DATA ENDS
; ZZZZ is the code table for the keyboard.

STACK-SEG SEGMENT
DW 100 DUP(0)
T-STACK LABEL WORD
STACK-SEG ENDS
;
CODE SEGMENT
ASSUME CS: CODE, DS: DATA, SS: STACK-SEG
;
; Initialize the segment registers.
;
START:           MOV AX, DATA
                MOV DS, AX
                MOV AX, STACK-SEG
                MOV SS, AX
                MOV SP, OFFSET T-STACK
;
; Initialize Intel 8279.
;
                MOV AL, 00H
                OUT X1, AL
                MOV AL, YY ; Program clock
                OUT X1, AL
;
; Insert the address of ISR KBDP in interrupt vector table. Since
interrupt type is 20H, the
; Offset and Segment addresses are to be stored at 20H □ 4 = 0080H
as:
; 0080H, 0081H—Offset address and 0082H, 0083H—Segment
address
;

```

```

MOV AX, 0000H
MOV ES, AX
MOV WORD PTR ES: 0080H, OFFSET KBDP
MOV WORD PTR ES: 0082H, SEG KBDP

;

; Program
;

—
—
—
;

; ISR KBDP Procedure
;

KBDP PROC NEAR
MOV AL, YY ; Program clock
OUT X1, AL

;

; Send command to Intel 8279 to read FIFO from 00 location—auto
increment.
; Command = 0 1 0 1 0 0 0 = 50H
;

MOV AL, 50H
OUT X1, AL
IN AL, X0

;

; Find the code of the key pressed. The FIND procedure finds out the
code, stores in
; register AL and returns.
;

CALL FIND
;

; Process the code as required by the application.
;

CALL PROG
IRET
KBDP ENDP

;

; Procedure to find the code of depressed key
;

FIND PROC NEAR

```

```

;

; The scanned keyboard data is stored in register AL before invoking
this procedure.

; The format is: D7—CNTL, D6—SHIFT, D5 D4 D3—SCAN, i.e.
Row No.,
; D2 D1 D0—RETURN, i.e. Column No.

;

        LEA    BX, ZZZZ
        MOV    TEMP, AL
        AND    AX, 0007H ; Col. No. in register AX
        MOV    COL, AX
        MOV    AL, TEMP
        AND    AX, 0038H

; 8 □ Row No., i.e. the displacement from the start of the table to the
start of the row is
; placed in AX.

        MOV    ROW, AX
;

; Fetch keycode from table and store in register AL.
;

        ADD    BX, ROW
        ADD    BX, COL
        MOV    AL, (BX)
        RET

FIND    ENDP

PROG   PROC NEAR

;

; This procedure processes the code of the depressed key as required
by the application.

;

        —
        —
        RET

PROG   ENDP

CODE  ENDS

```

The 8086 program for Figure 7.49(b)—IRQ not used

```

;

; Intel 8086 software for the keyboard using the 8279 when IRQ is not
used.

```

```

;

; For data read A0 = 0 and RD = low
; For FIFO status read A0 = 0 and RD = low
; For command output to 8279 A0 = 1 and WR = low
; For data write to 8279 A0 = 0 and WR = low

DATA SEGMENT
    X0    EQU —; Port for Intel 8279 data
    X1    EQU —; Port for Intel 8279 command
    Y1    EQU —; FIFO status input port
    YY    DB —; Code to program clock
    ZZZZ  DB -, -, -, -, -, -, -, ; Codes for different
          keys
DATA ENDS

STACK SEGMENT
    DW 200 DUP(0)
    T-STACK LABEL WORD
STACK ENDS

CODE SEGMENT
    ASSUME CS: CODE, DS: DATA, SS: STACK
;

; Initialize segment registers and stack pointer.
;
    MOV    AX, DATA
    MOV    DS, AX
    MOV    AX, STACK
    MOV    SS, AX
    MOV    SP, OFFSET T-STACK
;

; Initialize Intel 8279.
;
    MOV    AL, 50H
    OUT    X1, AL
;

; Program Clock
;
    MOV    AL, YY
    OUT    X1, AL

LOOP1:   —

```

```
—  
—  
CALL KBDPG  
—  
—  
—  
JMP LOOP1  
;  
; Keyboard program  
;  
KBDPG PROC NEAR  
    LEA SI, ZZZZ  
; SI is used for indirect addressing of code table in procedure FIND  
    IN AL, Y1  
;  
; (AL) = FIFO Status, D2 D1 D0 = No. of Characters  
;  
    AND AL, 07H  
    JZ NEXT  
    MOV CL, AL ; (CL) = Count  
; Read FIFO from 00 location in auto-increment mode.  
; Command = 01010000 = 50H  
    LOOP2:    MOV AL, 50H  
              OUT X1, AL  
              IN AL, X0  
              CALL FIND  
              CALL PROG  
              DEC CL  
              JNZ LOOP2  
NEXT:      NOP  
          RET  
KBDPG ENDP  
FIND PROC NEAR  
—  
—  
RET  
FIND ENDP  
PROG PROC NEAR  
—
```

```

    —
RET
PROG ENDP
CODE ENDS

```

Figure 7.50 shows a flowchart for interface to 16-character LED display.

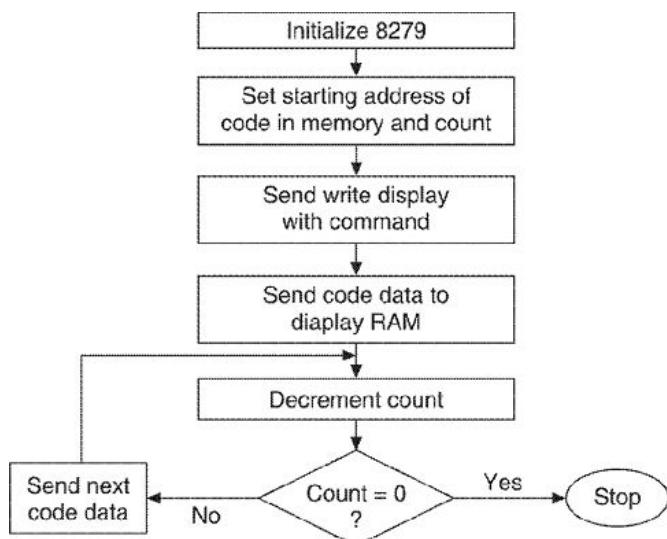


Figure 7.50 Flowchart to interface the 16-character LED display to microprocessor using the 8279 chip.

The interface programs for the 8085 and for the 8086 are given below.

The 8085 program for Figure 7.50

```

;
; Software to interface 16 character LED display to Intel 8085 using
Intel 8279
;
; For command output to 8279      A0 = 1 and WR = low
; For data write to 8279          A0 = 0 and WR = low

X1    EQU —; Command port
Y0    EQU —; Data output port
; Mode Set—16 8-bit character display—left entry
; Mode Word = 0 0 0 0 1 0 0 0 = 08H
        MVI   A, 08H
        OUT   X1
        LXI   H, ZZZZH
        MVI   C, 10H ; Total 16 bytes to be displayed.

;
; Write display RAM from 0000 location—auto-increment mode.

```

```

; Command = 1 0 0 1 0 0 0 = 90H
;
    MVI   A, 90H
    OUT  X1
LOOP:           MOV   A, M
    OUT  Y0
    INX  H
    DCR  C
    JNZ  LOOP
    —
    —
    —
    HLT
ZZZZ           DB  -, -, -, -, -, -, — ; Bytes to be
displayed

```

The 8086 program for Figure 7.50

```

; For command output to 8279 A0 = 1 and WR = low
; For data write to 8279 A0 = 0 and WR = low
; Software to interface 16 character LED display to Intel 8086 using
Intel 8279
;
; Mode Set—16 8-bit character display—left entry
; Mode Word = 0 0 0 0 1 0 0 0 = 08H
DATA SEGMENT
    X1      EQU —; Command port
    Y0      EQU —; Data output port
    DISP-CODE DB —, —, —, —, —, —, —, — ; Display codes of
characters
DATA ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA
    MOV     AX, DATA
    MOV     DS, AX
    LEA     SI, DISP-CODE
    MOV     AL, 08H
    OUT     X1, AL
    MOV     CX, 10H ; Total 16 bytes to be displayed.
;

```

```

; Write display RAM from 0000 location—auto-increment mode.
; Command = 1 0 0 1 0 0 0 = 90H
;
    MOV  AL, 90H
    OUT X1, AL
LOOP1:   MOV  AL, (SI)
    OUT Y0, AL ; A0 = 0, i.e. Data
    INC  SI
    LOOP LOOP1
    —
    —
    —
    HLT
CODE ENDS

```

EXAMPLE 7.4

A metro railway station has a number of self-operated ticket purchase counters. The passenger is required to enter the codes of the starting station and the destination station. The microprocessor calculates the fare and it is displayed on four seven-segment LEDs, in the following format:



The passenger deposits the amount through the slot and gets the printed ticket. Let us design a system to perform the above functions till the display of the fare.

Solution:

Let us assume that there are 64 stations in the metro which are pre-codified and each key in the 8 × 8 keyboard is dedicated to a station. Thus, if a passenger presses the key number 7 followed by the key number 20, the fare between the station number 7 and the station number 20 is calculated and displayed.

The keyboard is arranged in the following fashion:

Stn 1	Stn 2	Stn 8
Stn 9	Stn 10	Stn 16
Stn 57	Stn 58	Stn 64

Thus row no. j and column no. k will mean Stn no.— $8 \times (j - 1) + k$. You may assume any other way of arranging this and consequently find the station number.

The 2-byte display code, ST–CD, for the stations is pre-stored in the memory array as follows:

Display code for Stn 1
Display code for Stn 2

Following is the layout of seven-segment LED display for passengers:

Starting Rupee t)	Stn Fare (2 digit) (2 digit)	Code Paise	Destination Stn (2 digit)	Code (2 digit)	Fare \overline{WR}

Let us assume the following port \overline{RD} ddresses:

X1 – Command to 8279 ($A_0 = 1$ and $\overline{WR} = \text{low}$)

X0 – Data read from 8279 ($A_0 = 0$ and $= \text{low}$)

Y0 – Data write to 8279 ($A_0 = 0$ and $= \text{low}$)

Now let us discuss the major steps in this application:

1. Programming the keyboard/display mode

Keyboard: Since it is an 8 \square 8 keyboard, encoded scan will be required. At the same time, 2-key lockout for debouncing will be appropriate. Thus,

$$KKK = 000$$

Display: Eight numbers of 8-bit character display are used. The left entry mode will be appropriate considering that the starting Stn code will be entered first and it occupies the leftmost position. Thus,

$$DD = 00$$

Hence, the keyboard/display mode command = 0000 0000 = 00H

2. Program clock

To obtain 100 kHz internal frequency, the external clock frequency must be divided by suitably programming the clock. Thus, the 8279 may be initialized in this way.

3. Operation

When a passenger presses a key, it causes an interrupt in the microprocessor since the IRQ line of the 8279 is connected to an interrupt pin of the microprocessor.

The first key pressed interrupt will be for the starting station and the second key pressed interrupt will be corresponding to the destination station. The calculation of the fare and the display of the station numbers and the fare will occur only after the second key is pressed.

Let us define a variable KPRSD (for the key pressed status)

when $KPRSD = 0 \rightarrow \square$ No key pressed

$= 1 \rightarrow \square$ First key pressed

$= 2 \rightarrow \square$ Second key pressed

Now let us design the systems for this application using the 8085 and 8086 microprocessors.

Using the 8085

Figure 7.51 shows the 8085 interfaced to an 8 \times 8 key keyboard and eight seven-segment LEDs through the 8279. The IRQ output of the 8279 is connected to the RST 5.5 interrupt pin of the 8085.

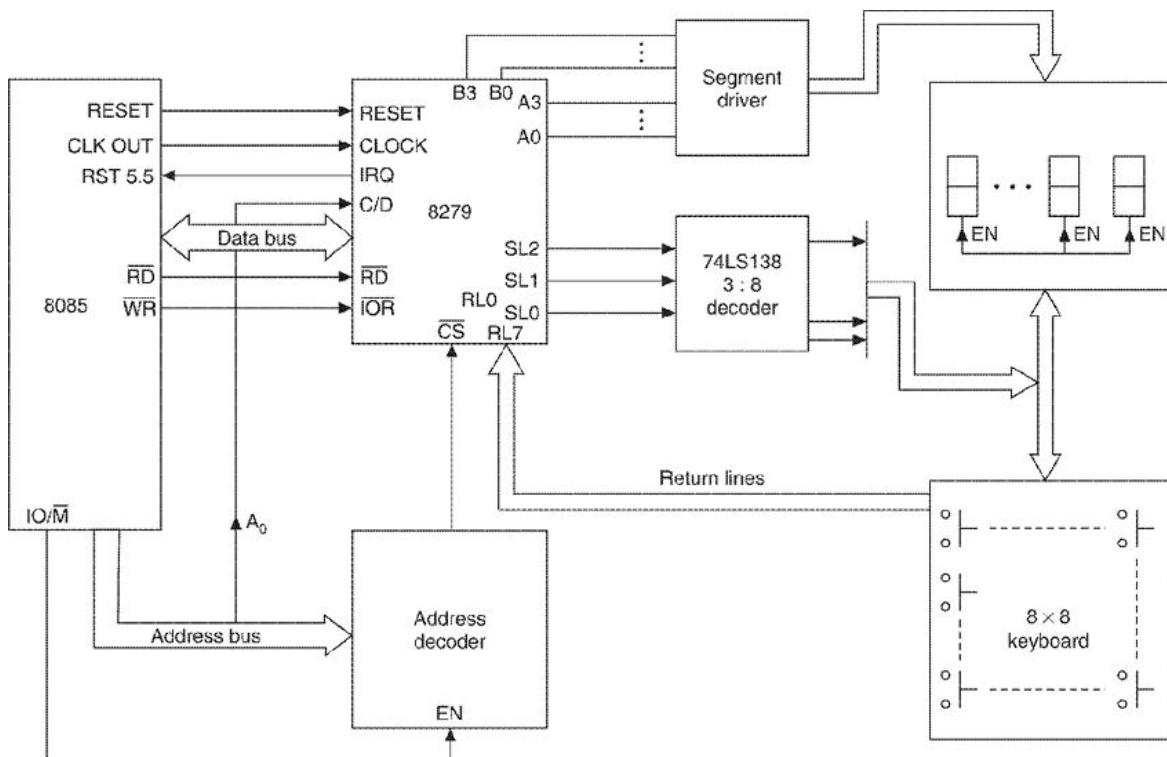


Figure 7.51 Schematic diagram of the 8085-based automatic ticket counter at a metro railway.

; Main Program

ST-CD	DB	-,-,-,-; Display codes for stations
DIS-CD	DSB	8 ; Display codes for Intel 8279
TEMP	DB	0
START-STN	DB	0
DEST-STN	DB	0

```

X1      EQU  — ; Command port
X0      EQU  — ; Data input port
Y0      EQU  — ; Data input port
; Initialization of Intel 8279
; Set keyboard/display mode.
;
    MVI   A, 00H
    OUT   X1
;
; Program Clock
; Assume that the CLK OUT frequency is 2 MHz. To obtain 100 kHz
internal
; frequency, PPPPP = 10100. Thus, the program clock command =
00110100 = 34H.
    MVI   A, 34H
    OUT   X1
    ISR   RST 5.5
; Check and update the KPRSD variable; assume that KPRSD is
represented in register D.
    MOV   A, D
    CPI   01H
    JM    FIRST
    JZ    SECND
    JMP   ERROR
FIRST:   MVI   D, 01 ; Starting station key pressed
    JMP   NEXT
SECND:   MVI   D, 02 ; Destination station key pressed
    JMP   NEXT1
; Read FIFO from 00 location—auto-increment command = 0101
0000 = 50H
NEXT:    MVI   A, 50H
    OUT   X1
NEXT1:   IN    X0 ; ACC contains the FIFO location
content.
    CALL  FIND ; Find the station number.
;
; Store the station number in memory START-STN or DEST-STN
location.
;

```

```

STA    TEMP ; Store the station no. in TEMP.
MOV    A, D
CPI    01H
JZ     FIRST-STN
JP     SCND-STN
JM     ERROR
FIRST-STN:   LDA    TEMP
              STA    START-STN
              EI    ; Enable interrupt system for future interrupts.
              RET

SCND-STN:    LDA    TEMP
              STA    DEST-STN
;
; Calculate the fare based on START-STN and DEST-STN.
;
CALL  CLFARE
;
; Fare calculated is in RFARE1, RFARE2 (rupee value) and PFARE1,
PFARE2 (paise
; value) in BCD code, fare of Rs. 35.75 will be represented as
; 0.....0011      0.....0101      0.....0111      0.....0101
; RFARE1        RFARE2        PFARE1        PFARE2
;
; Assemble the byte code to be displayed in display RAM at DIS-CD.
;
CALL  DISP-ASB
; Display station numbers and fare.
LXI    H, DIS-CD
MVI    C, 08H ; 8 bytes to be displayed
; Write display RAM from 0000H location—auto-increment mode.
; Command = 1001 0000 = 90H
MVI    A, 90H
OUT   X1
DISP:   MOV    A, M
OUT   Y0
INX   H
DCR   C
JNZ   DISP
; Reset KPRSD variable.

```

```
MVI D, 00  
ERROR: EI  
RET
```

;

Subroutine FIND
; To find the station number

7	6	5	4	3	2	1	0
CNTL	SHIFT	SCAN					RETURN

```
MOV B, A ; Move the scanned keyboard data to register B  
ANI 07H ; Column No. in register A  
MOV C, A ; (C) □ Column No.  
MOV A, B ; (A) □ Scanned keyboard data  
ANI 38H ; (A) □ 8 □ Row number (38H = 00111000)  
SUI 08H ; (A) □ 8 □ (j - 1), where j = Row No.  
ADD C  
;  
;(A) □ 8 □ (j - 1) + k, where j = Row No. and k = Column No. Thus (A) □ STN. NO.  
;  
RET
```

Subroutine CLFARE

;
; Calculate the fare from the starting and the destination station. It can be stored in the
; look-up table or may be calculated based on some predefined formula.
;

Subroutine DISP-ASB

; Based on two station nos. and fare value in BCD, the display byte codes can be

; assembled at DIS-CD.

;

; For START_STN

;

```
LXI H, ST-CD  
LDA START-STN  
SUI 01H; (A) □ START-STN—1  
RAL ; (A) □ (A) □ 2
```

```

ADD  L
MOV  L, A ; (H) (L) □ Starting address of display code
MOV  B, M
INX  H
MOV  C, M
LXI  H, DIS-CD
MOV  M, B
INX  H
MOV  M, C
;
; Similarly for DEST_STN
:
:
:
:
; Find the display code for fare value in BCD form and store.
:
:
:
RET

```

Using the 8086

Figure 7.52 shows the 8086 interfaced to an 8 × 8 key keyboard and eight seven-segment LEDs through the 8279. The IRQ output of the 8279 is connected to the INTR pin of the 8085, and it generates type 48, i.e. 30H interrupt.

The 8279 requires internal clock frequency to be 100 kHz and the clock input must be divided through Program Clock Command. Consider that the clock frequency of the 8086 is 5 MHz. This cannot be connected directly to the 8279 as it cannot be converted to 100 kHz by performing division by maximum 31. Therefore, we must divide the clock externally, using the divider or using the PCLK output of the 8284 clock generator. Thus, for 5 MHz clock frequency, the PCLK frequency is 2.5 MHz, which must be divided by 25 to get 100 kHz.

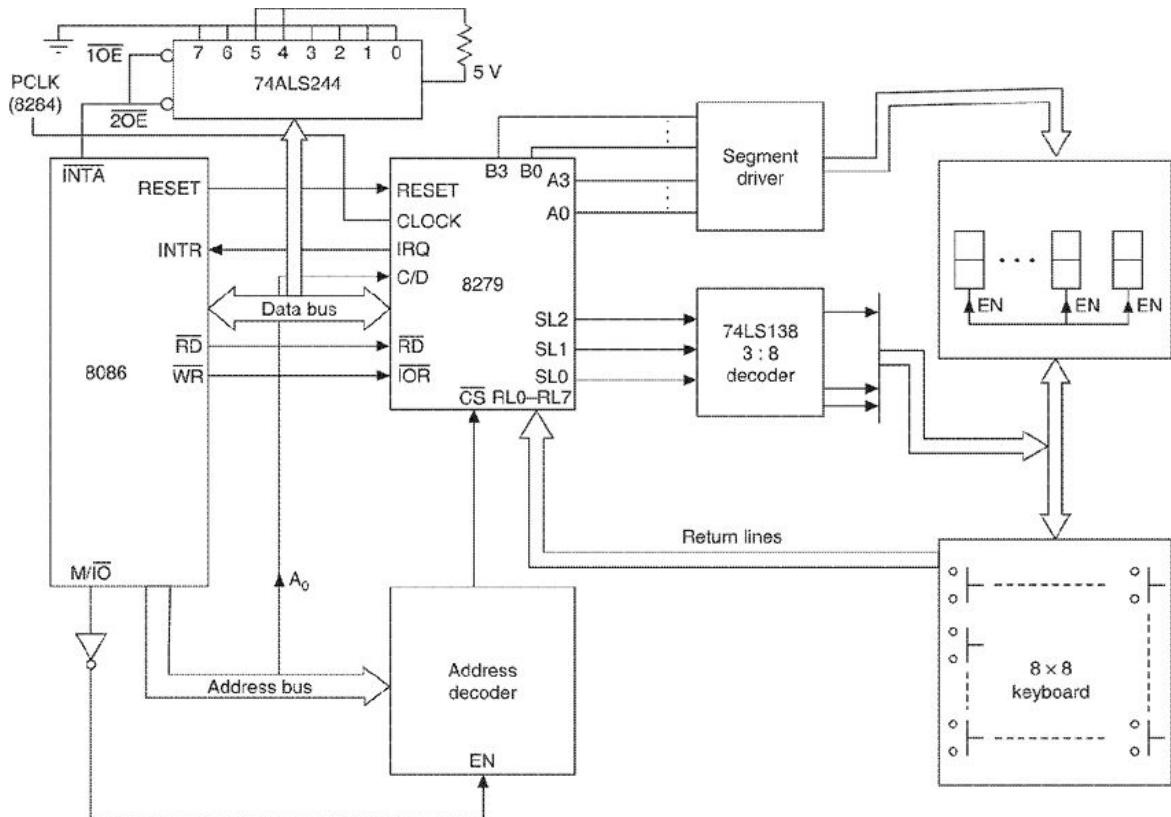


Figure 7.52 Schematic diagram of the 8086-based system for automatic ticket counter at a metro railway station.

; Intel 8086 program for an automatic ticket machine at a metro railway station,

```

;
;
; Main Program
;
DATA SEGMENT
    ST-CD        DW      -, -, -, -, -, --; Display code
    for
; stations
    DIS-CD      DB      8 DUP (0) ; Codes for
    display
    KPRSD       DB      0 ; Variable keypressed
    X1          EQU     —; Command port
    X0          EQU     —; Data input port
    Y0          EQU     —; Data input port
    START-STN   DSB     1
    DEST-STN   DSB     1
    TEMP        DB      0
    COL         DB      0

```

```

        ROW          DB    0
        BCODE1       DB    0
        BCODE2       DB    0
DATA ENDS
S-STACK      SEGMENT
        DW    100 DUP(0)
T-STACK      LABEL WORD
S-STACK      ENDS
;
CODE SEGMENT
ASSUME CS: CODE, DS: DATA, SS: S-STACK
;
; Initialize the segment registers.
;
START:      MOV   AX, DATA
            MOV   DS, AX
            MOV   AX, S-STACK
            MOV   SS, AX
            MOV   SP, OFFSET T-STACK
;
; Initialize Intel 8279.
;
            MOV   AL, 00H
            OUT   X1, AL
; Assume that Intel 8086 frequency is 5 MHz. Thus PCLK frequency
is 2.5 MHz.
; It is divided by 25 by program clock command to get 100 kHz.
Thus,
; Program Clock Command = 0 0 1 1 1 0 0 1 = 39H.
;
            MOV   AL, 39H
            OUT   X1, AL
;
; Insert the address of ISR KPIRS in the interrupt vector table. Since
the interrupt type is
; 30H, the offset and segment addresses are to be stored at 30H □ 4 =
00C0H as—Offset
; address at 00C0H, 00C1H and Segment address at 00C2H, 00C3H.
;
            MOV   AX, 0000H

```

```

        MOV  ES, AX
        MOV  WORD PTR ES: 00C0H, OFFSET KPIRS
        MOV  WORD PTR ES: 00C2H, SEG KPIRS
;
; Program
;
;

;
; ISR KPIRS
;
        KPIRS      PROC   NEAR
        MOV  AL, KPRSD
        CMP  AL, 01H
        JL   FIRST
        JZ   SCND
        JMP  ERROR
;
; Starting station key pressed
;
        FIRST:      MOV   KPRSD, 01H
        JMP  NEXT
;
; Destination station key pressed
;
        SCND:       MOV   KPRSD, 02H
        JMP  NEXT1
;
; Read FIFO from 00 location—auto increment.
; Command = 0 1 0 1 0 0 0 = 50H
;
        NEXT:       MOV   AL, 50H
        OUT  X1, AL
        NEXT1:      IN    AL, X0 ; (AL) = FIFO location
contents
        CALL  FIND  ; Find station no.
;
; Store station no. as memory variables START-STN or DEST-STN.
;

```

```

        MOV    TEMP, AL
        MOV    AL, KPRSD
        CMP    AL, 01H
        JZ     FIRST-STN
        JA     SCND-STN
        JL     ERROR

FIRST-STN:      MOV    AL, TEMP
                MOV    START-STN, AL
;
; Enable interrupt system for future interrupts.
;
        EI
        JMP    FINI

SCND-STN:      MOV    AL, TEMP
                MOV    DEST-STN, AL
;
; Calculate the fare based on START-STN and DEST-STN.
;
        CALL   CLFARE
;
; Fare calculated is in the format—RAFARE1, RAFARE2 (Rupee
value) and PFARE1,
; PFARE2 (Paise value) in BCD code. Fare of Rs. 35.75 will be
represented as—
; 00....0011      00...0101      00...0111      00...0101
; RFARE1         RFARE2        PFARE1        PFARE2
; Assemble the byte code to be displayed in RAM as DIS-CD.
;
        CALL   DISP-ASB
;
; Display station numbers and fare.
;
        LEA    DI, DIS-CD
        MOV    CX, 08H
;
; Write display RAM from 00 location—auto increment.
; Command = 1 0 0 1 0 0 0 = 90H
;
        MOV    AL, 90H
        OUT   X1, AL

```

```

DISP:           MOV   AL, (DI)
                OUT   Y0, AL
                INC   DI
                LOOP  DISP
;
; Reset KPRSD variable.
;
MOV   KPRSD, 00H
ERROR:          EI
FINI:           NOP
IRET
KPIRS ENDP
FIND  PROC  NEAR
;
; To find station no.
; The format of scanned keyboard data is D7—CNTL, D6—SHIFT
; D5 D4 D3—SCAN, i.e. Row No., D2 D1 D0—RETURN, i.e.
Column No.
;
MOV   BL, AL
AND   AL, 07H ; (AL) = Column No.
MOV   COL, AL
AND   BL, 38H ; (BL) = 8 □ Row No.
MOV   ROW, BL
SUB   BL, 08H ; (BL) = 8 □ (j – 1), j = Row No.
ADD   BL, COL
;
; (BL) = 8 □ (j – 1) + k, where j = Row No. and k = Column No. Thus
(BL) = Station No.
MOV   AL, BL
RET
FIND  ENDP
CLFARE      PROC  NEAR
;
; Calculate the fare from the starting to the destination station. The
fare may be stored as a
; look-up table or may be calculated based on some pre-defined
formula.
;

```

```

        RET
CLFARE      ENDP
DISP-ASB     PROC NEAR
;
; Based on two station numbers and fare value in BCD, the display
byte code may be
; assembled in DIS-CD in memory.
;
;
;
; For START-STN
;
;
; Two byte display code of stations are pre-stored in memory at ST-
CD
;
LEA    BP, ST-CD
MOV    AL, START-STN
DEC    AL ; (AL) = START-STN - 1
SAL    AL
ADD    BP, AX ; (BP) = Starting address of display
code
MOV    AL, (BP)
MOV    BCODE1, AL
INC    BP
MOV    AL, (BP)
MOV    BCODE2, AL
LEA    BX, DIS-CD
MOV    (BX), BCODE1
INC    BX
MOV    (BX), BCODE2
;
; Similarly for DEST-STN
;
—
—
—
;
; Find the display code for fare value in BCD form and store.
;
—
—

```

```
—  
    RET  
DISP-ASB      ENDP  
    CODE  ENDS
```

EXERCISES

1. Redesign the system of Example 7.4 and develop software for both the 8085 and the 8086 microprocessors to have rupee part of the fare in three digits like Rs. 524.77.
2. Redesign the systems in Exercise 1 above for 128 stations for both the 8085 and 8086 microprocessors.

7.7 PROGRAMMABLE INTERVAL TIMERS INTEL 8253 AND INTEL 8254

One of the most common problems in any microcomputer system is the generation of accurate time delays under software control. We have seen so many applications where the processor is made to wait for sometime. For example, in the keyboard interface, the CPU waits about 5 ms in case of each key depression, in order to check for false depressions.

The common software approach is to set up the timing loop by moving a fixed number to a register pair and decrementing that number till it becomes zero. The fixed number depends on the time delay required. This approach loads the CPU unnecessarily.

The 8253 solves this problem by facilitating three 16-bit programmable counters on the same chip. The programmer configures the 8253 to match his requirements and initializes one of the counters with the desired quantity. The 8253, on receiving the command from the CPU, counts out the delay and interrupts the CPU at the end, thus reducing the software overheads. Other functions which can be achieved by the 8253, are as follows:

- Event Counter
- Programmable Rate Generator
- Binary Rate Multiplier
- Real Time Clock
- Digital One-Shot
- Complex Motor Controller

The functional block diagram of the 8253 is shown in Figure 7.53. Each counter consists of a single, 16-bit programmable down counter which can operate either in binary or in BCD. Each counter has two inputs as CLK and GATE and one output as OUT. The input clock is fed to CLK. For the 8253 the clock frequency should be 2 MHz or less. If the clock frequency of the microprocessor is more than 2 MHz, it must be suitably reduced by the division.

The 8254 handles signals from dc to 10 MHz frequency.

- | | | |
|--------------|---|--------------|
| Up to 8 MHz | — | Intel 8254 |
| Up to 10 MHz | — | Intel 8254-2 |

The 8254 is the superset of the 8253.

The functions of GATE and OUT depend on the mode, in which the particular counter is set into.

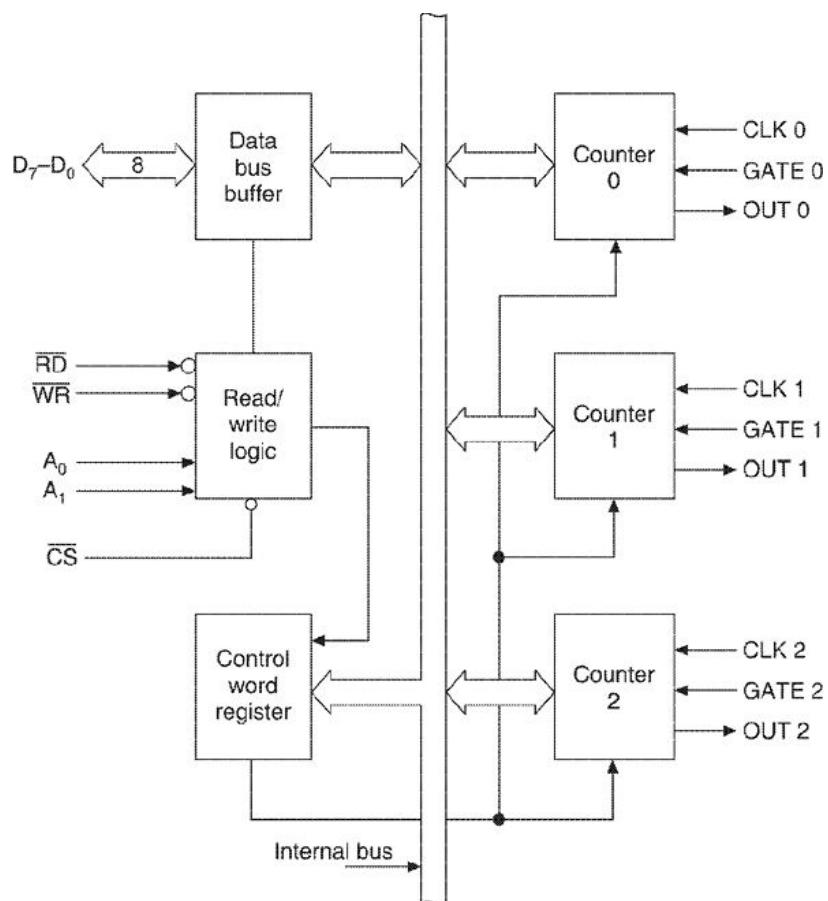


Figure 7.53 Functional block diagram of the 8253 chip.

7.7.1 Operating Modes

The counters are fully independent and each can have separate mode configurations and counting operations, binary or BCD. Following are the different modes possible in the 8253/8254.

Mode 0—interrupt on terminal count

This mode is typically used for event counting. The GATE input is used to enable or disable the counting. When the GATE input is high, the counting is enabled and when it is low the counting is disabled. After the count is loaded into the selected count registers, the OUT remains low and the counter starts counting one cycle after the counter is loaded. On reaching the terminal count (i.e. zero), OUT will go high and remain high until the count register is reloaded or the control word is written. The OUT point can be connected to any interrupt request pin of the microprocessor.

Mode 1—programmable one-shot

The GATE input is used to trigger the OUT. The OUT will be initially high and go low on the count following the rising edge at the GATE. When the terminal count is reached, the OUT goes high and remains high until the CLK pulse after the next trigger. Thus, the duration of one-shot pulse at OUT can be programmed through the count.

Mode 2—rate Generator

The counter in this mode acts as divide-by-N counter. It is used to generate a real-time clock interrupt. The OUT will initially be high. When the count has decremented to 1, the OUT goes low for one clock pulse. The OUT then goes high again, the counter reloads the initial count and the process is repeated. The GATE input, when low, disables the counting. When the GATE input goes high, the counter starts from the initial count. The OUT gives one pulse after every N clock pulses, thus achieving the rate generator function.

Mode 3—square wave generator

It is similar to Mode 2 except that the OUT remains high for the first-half of the count and goes low for the other-half, thus generating a programmable square wave shape at OUT. Suppose n is the number loaded into the counter, then the OUT will be

- High for $n/2$ counts and low for $n/2$ counts if $n = \text{even}$
 - High for $(n + 1)/2$ counts and low for $(n - 1)/2$ counts if $n = \text{odd}$.
- The GATE input function is the same as that in Mode 2.

Mode 4—software triggered strobe

The OUT is initially high and goes low for one clock period on the terminal count. Thus, a pulse can be generated by software. The GATE

input when low disables the counting and when high, enables the counting. The difference between Mode 4 and Mode 2 is that in Mode 2, the OUT pulses are generated continuously after every N clock pulses (i.e. on terminal count but N is reloaded automatically), but in Mode 4, the OUT pulse is generated only once after N clock pulses (i.e. terminal count, but no automatic reloading).

Mode 5—hardware triggered strobe

The OUT is initially high. The counter will start counting after the rising edge of the GATE input and the OUT will go low for one clock period when the terminal count is reached.

The hardware circuit should trigger the GATE input to initiate the counting operation, thereby generating the OUT pulse.

The operation of the GATE input in different modes is summarized in Figure 7.54.

Mode	Low or going low	Rising	High
0	Disables counting	—	Enables counting
1	—	(1) Initiates counting (2) Resets output after the next clock	—
2	(1) Disables counting (2) Sets the output immediately high	Initiates counting	Enables counting
3	(1) Disables counting (2) Sets the output immediately high	Initiates counting	Enables counting
4	Disables counting	—	Enables counting
5	—	Initiates counting	—

Figure 7.54 Operation of the GATE input.

7.7.2 Programming the 8253

Just like the 8255, in the 8253 too, the control words are used to program the counters with the desired mode and quantity information. These control words are sent to the Control Word Register for each counter to be programmed. The format of the control word is as follows:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

- (a) The SC1 and SC0 bits select one of the three counters of the 8253 (SC1, SC0 = 00—counter 0, 01—counter 1, 10—counter 2, 11—illegal)

In case of the 8254, when (SC1, SC0 = 11) then it is a read back command which is different from that of the 8253.

- (b) The M2, M1 and M0 bits select the mode (M2, M1, M0 = 000—Mode 0, 001—Mode 1, X10—Mode 2, X11—Mode 3, 100—Mode 4, 101—Mode 5), X = Don't care
- (c) The BCD bit selects the BCD counting or the Binary counting (BCD = 0—Binary counting 16-bit, 1—Binary coded decimal (BCD) counter (4 decades))
- (d) The RW1, RW0 bits identify the Read/Write operation for the count as follows:

RW1 RW0

- | | |
|----------|--|
| 0 0 | — Counter latching operation. Used when the programmer wants to read the contents of any counter “on the fly”, i.e. without disturbing the counting. |
| 0 1 | — Read/Write the least significant byte only. |
| 1 0 | — Read/Write the most significant byte only. |
| 1 1 | — Read/Write the least significant byte first and then the most significant byte. |

The actual order of programming is quite flexible. The control words can be written in any sequence of the counter selection. The loading of the count register with the actual count value, however, must be done in the actual sequence specified by the RW0 RW1 bits in the control word for any counter. That means:

- (a) For each counter, the control word must be written before the initial count is written.
- (b) The initial count must follow the count format, i.e. RW1, RW0 = 01—least significant byte only, RW1, RW0 = 11—least significant byte followed by the most significant byte.

If a counter is programmed to read/write two byte counts, it must be ensured that the program does not transfer the control between writing the first byte and the second byte to another routine which also writes

into the same counter. Otherwise, an incorrect value will get loaded to the counter.

The contents of the counter can be read by a simple I/O read operation. However, counting should be inhibited before this, by the GATE input. The other method is the read “on the fly” (also known as Counter Latch Command), i.e. read without disturbing the counting process. To achieve this, the control register is loaded with a special control word as shown below:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
SC1	SC0	0	0	X	X	X	X

The SC1 and SC0 bits specify the counter. The contents of the counter are latched into a storage register associated with the counter in the 8253. Using a simple I/O read operation, the storage register, i.e. the contents of counter, can be read.

Another possible method for reading the counter is to use the Read Back command. This is specific to the 8254 and is not present in the 8253. By this command, the user can check the count value, the programmed mode and the current state of the OUT pin and the NULL count flag of the selected counters. The Read Back command format is shown in Figure 7.55.

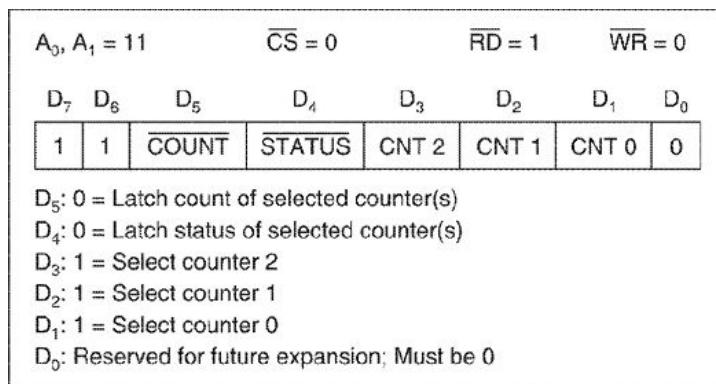


Figure 7.55 Read Back command format.

The command is written into the control word register. The Read Back command may be used to latch the outputs of either one or more counters by setting the COUNT bit D₅ = 0 and selecting the desired counters. This single command is functionally equivalent to multiple Counter Latch commands. The latched count of each counter is held until it is read or the counter is reprogrammed.

7.7.3 Interfacing the 8253/8254

Figure 7.56 shows the interfacing of the 8085 with the 8253.

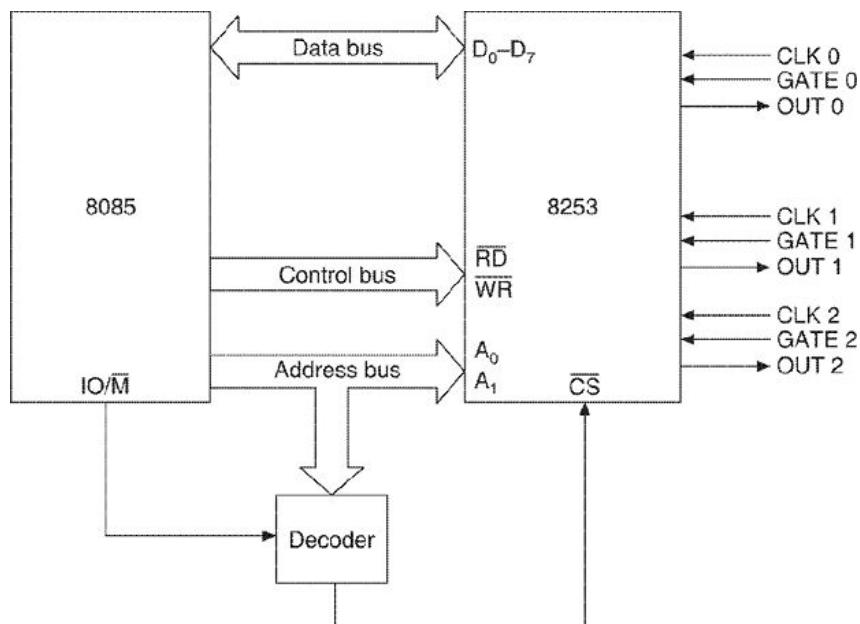


Figure 7.56 Interfacing the 8085 with 8253/825.

The interfacing to the 8086 in minimum mode is shown in Figure 7.57. The AD₀–AD₇ lines are connected to D₀–D₇.

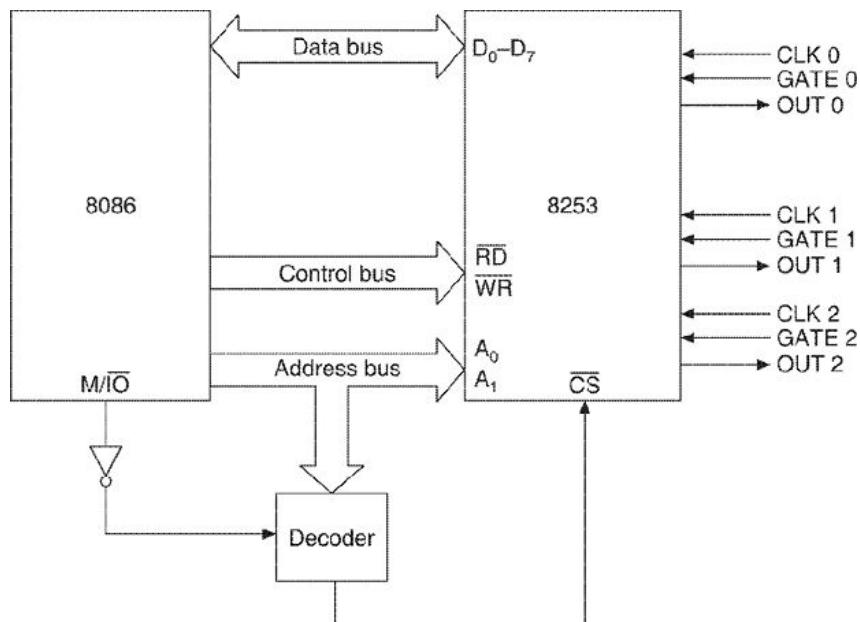


Figure 7.57 Interfacing the 8086 with 8253/8254.

The interfacing of the 8254 to both the 8085 and the 8086 are exactly the same. Since A₀ and A₁ address lines are connected to the 8253, the counters/control word are defined with respect to the values of A₀ and A₁ as shown below.

A ₁	A ₀	
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word

Let us assume that the 8253 is selected when A₂ = 1 and A₃ = 1. Thus, the addresses for the different counters and the control words will be:

	A ₃	A ₂	A ₁	A ₀	
Counter 0	1	1	0	0	= 0CH
Counter 1	1	1	0	1	= 0DH
Counter 2	1	1	1	0	= 0EH
Control Word	1	1	1	1	= 0FH

The flowchart for the software for the 8253/8254 operation in Mode 0 (Interrupt on terminal count) consists of only two steps: (i) Initialize the 8253 and (ii) Load count. In the following we translate these steps in the programs of the 8085 and 8086 microprocessors.

Using the 8085

```

;
; The 8085 software for the 8253/8254 counter operation in Mode 0
;
; Circuit connections-
; A0 and A1 address lines of the 8085 are connected to A0 and A1
pins of the
; 8253/8254.
; The 8253/8254 is selected when A2 = High.
; Thus port nos. for the 8253/8254 are 04, 05, 06 and 07.
;
; Counter 0—Mode 0, Binary Mode, Count = FFDDH
; Control Word = 0 0 1 1 0 0 0 0 = 30H
; Initialize the 8253/8254.
;
        MVI   A, 30H
        OUT  07
        MVI   A, DDH
        OUT  04 ; LSB sent
        MVI   A, FFH ; MSB sent

```

```
OUT    04
```

```
—  
—  
—  
—
```

Using the 8086

```
;  
; The 8086 software for the 8253/8254 counter operation in Mode 0  
;  
; Circuit connections—  
; A0 and A1 address lines of the 8086 are connected to A0 and A1  
pins of the  
; 8253/8254.  
; The 8253/8254 is selected when A2 = High.  
; Thus port nos. for the 8253/8254 are 04, 05, 06 and 07.  
;  
; Counter 0—Mode 0, Binary Mode, Count = FFDDH  
; Control Word = 0 0 1 1 0 0 0 0 = 30H  
; Initialize the 8253/8254.  
;  
        DATA SEGMENT  
        CWORD      EQU      30H  
        COUNT-H    DB       FFH  
        COUNT-L    DB       DDH  
        DATA ENDS  
        CODE SEGMENT  
        ASSUME CS: CODE, DS: DATA  
        MOV AX, DATA  
        MOV DS, AX  
        MOV AL, CWORD  
        OUT 07H, AL  
        MOV AL, COUNT-L  
        OUT 04H, AL ; LSB sent  
        MOV AL, COUNT-H  
        OUT 04H, AL ; MSB sent
```

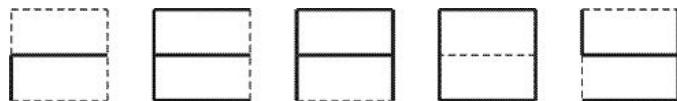
```
—  
—  
—  
—
```

CODE ENDS

EXAMPLE 7.5

In Example 7.4 concerning the 8279 interfacing, if a passenger enters the Starting Station Code and then changes his mind and leaves, the system will hang as it will wait for the Destination Station Code to be entered. Let us modify the design so that:

- (i) If the passenger has entered the Starting Station Code and there is no keyboard entry for the next 15 seconds, the system should reset.
- (ii) After the display of the Station Numbers and the Fare, the system should wait for 30 seconds and then only reset.
- (iii) The system should display rEADY initially and after reset. The display must be in the following fashion:



Solution:

Let us solve this problem by first using the 8085 and then by using the 8086 microprocessor.

The 8085-based design

Figure 7.58 shows the 8085 interfaced to the 8279 and the 8253. The 8279, in turn, is interfaced to eight seven-segment LEDs and a 8×8 keyboard as in the case of the previous example.

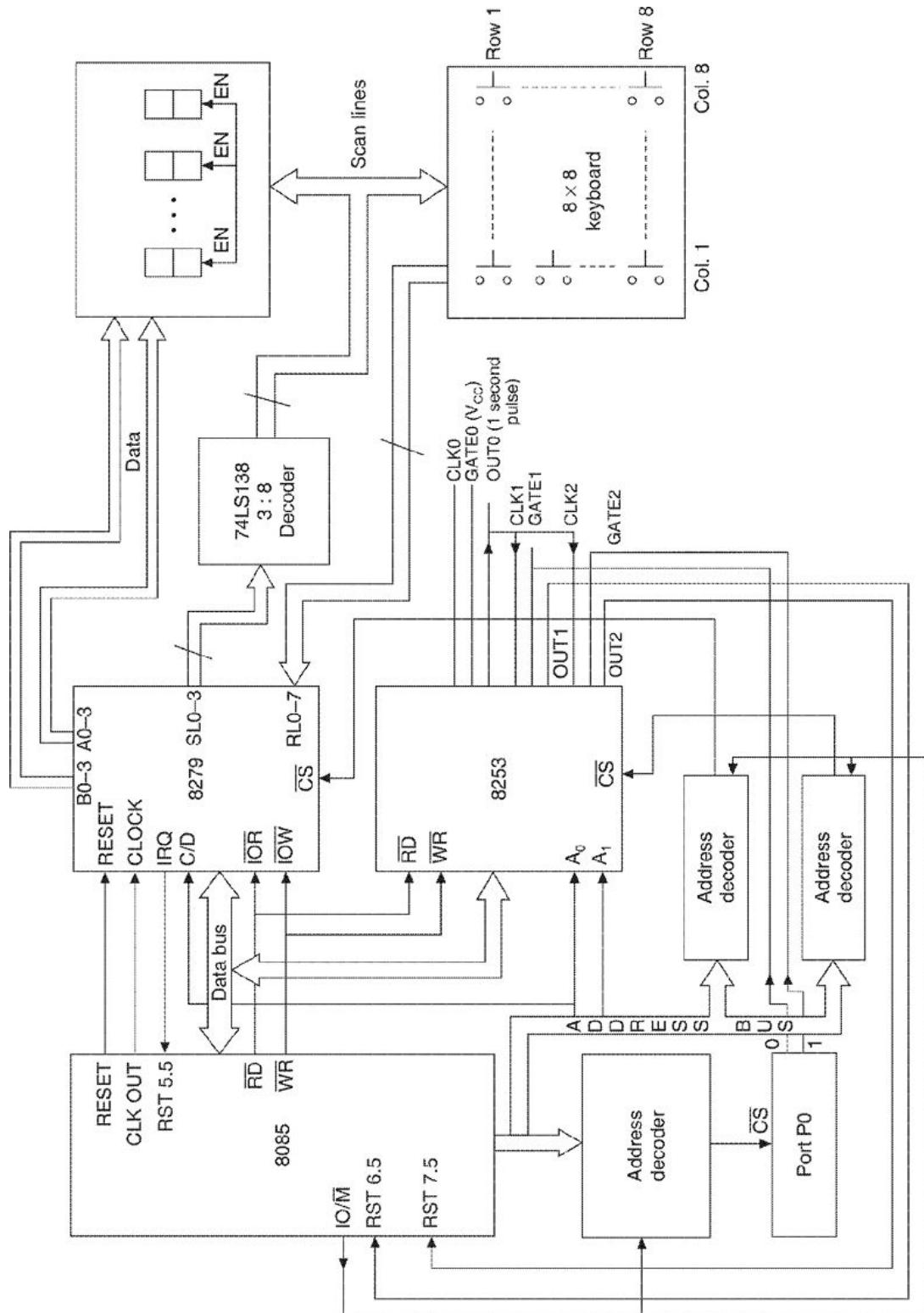


Figure 7.58 Schematic circuit diagram of the 8085-based system (with the 8253) for automatic ticket counter at a metro railway station.

We require two types of resets—a 15 second reset and a 30 second reset. Let us allocate the three timers/counters of the 8253 in the following manner:

Counter 0 — For production of 1 second clock pulse which will be input to Counter 1 and Counter 2.

Counter 1 — For 15 second reset.

Counter 2 — For 30 second reset.

The outputs of Counter1 and Counter 2, i.e. OUT1 and OUT2 have been connected to RST 6.5 and RST 7.5 respectively. The ISR for RST 6.5 and RST 7.5 will perform the necessary tasks of resetting. Let us assume the following port addresses for the 8253/54.

Counter 0	—	0CH
Counter 1	—	0DH
Counter 2	—	0EH
Control Word	—	0FH

Counter 0: Let us assume that the input clock frequency of the microprocessor is 4 MHz. The CLK OUT (2 MHz) of the 8085 is divided by 256 using two 7493s to get 7.8125 kHz clock (Figure 7.59). The time period of the clock is 0.128 millisecond. This is fed as CLK0 to Counter 0 of the 8253 to derive 1 second clock pulse at OUT0.

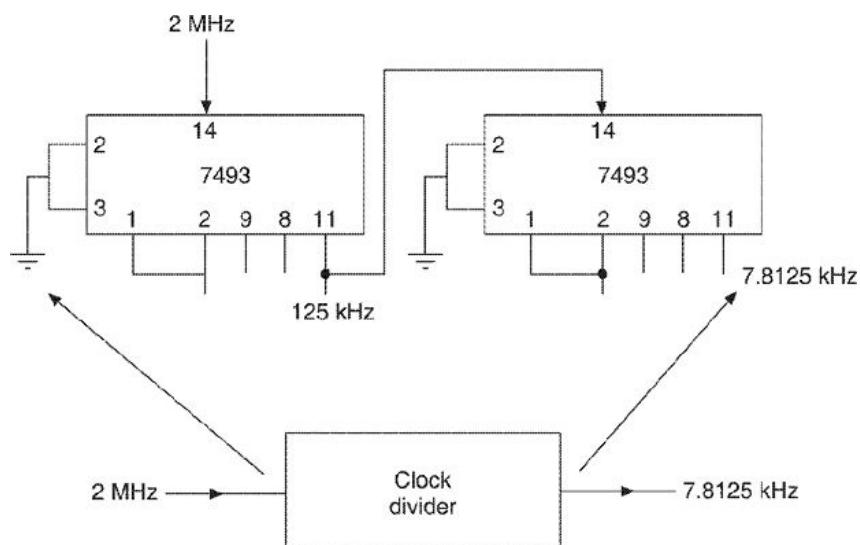


Figure 7.59 Generation of clock for counter.

The counter is used to generate 1 second pulse. The mode is therefore 2, i.e. the rate generator. It is selected as binary counter and both the lower-order and higher-order bytes of the count value are loaded into the counter. For 0.128 millisecond clock, the count value will be 7812. It will be represented as 1E84H.

The control word will be:

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

$$= 34H$$

Since the counter is working continuously, V_{CC} is connected to GATE0.

Counter 1: Counter 1 is used to generate the 15 second pulse to reset the system, if no key has been pressed in the last 15 seconds. The OUT1 output line has been connected to RST 6.5, and ISR 6.5 will perform the required functions. However, if a key has been pressed for Destination Station Code, the interrupt should not then occur, i.e. the interrupt should be disabled by the RST 5.5 ISR when the destination key press is detected. The mode will be 0, i.e. interrupt on the terminal count. It is the binary counter and since the count is only 15, only the least significant byte needs to be loaded.

Thus the control word will be:

0	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

$$= 50H$$

The counter will be started as soon as the first key press is detected. Thus, the high level at GATE1 needs to be programmed. It has been achieved through a port P0 in the figure. The P0.0 and P0.1 lines of P0 are connected to GATE1 and GATE2 respectively.

Thus, when 01 is written to Port P0 using

```
MV1 A, 01H
OUT P0
```

the GATE1 gets high level signal and the counter 1 gets activated.

Counter 2: Counter 2 is similar to counter 1 and is used to generate the 30 second pulse to reset the system, when the Station Numbers and the Fare have been displayed. The OUT2 output line has been connected to RST 7.5, and ISR will perform the required functions. The mode will be 0, i.e. interrupt on the terminal count. It is a binary counter and since the count is only 30, only the least significant byte needs to be loaded.

Thus the control word will be:

1	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---

$$= 90H$$

The counter will be started as soon as the Station Numbers and the Fare have been displayed. Like Counter 1, the activation of Counter 2 can also be programmed using Port P0. Since the P0.1 line is connected to GATE2, the following instruction will send a high-level signal at GATE2, thus activating Counter 2.

```
MVI    A, 02H
OUT    P0
```

Display: Let us now work out the display of rEADY on the seven-segment LED displays. A total of five seven-segment LEDs will be used in the following manner:

Letter	Segments	Display code byte		
r	e, g	1	0	1 0 1 1 1 1
E	a, d, e, f, g	1	0	0 0 0 1 1 0
A	a, b, c, e f, g	1	0	0 0 1 0 0 0
D	a, b, c, d, e, f	1	1	0 0 0 0 0 0
Y	b, c, d, f, g	1	0	0 1 0 0 0 1

The above display code bytes need to be loaded in the display RAM of the 8279 in sequence. Let us assume that the five display code bytes are stored in the consecutive memory locations as RD-CD and a subroutine RD-DISP will display the rEADY message when invoked.

Subroutine RD-DISP

; Let us erase, if any information was displayed previously using the display write
; inhibit/blanking command.

1	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

= A3H

; Both nibbles blanked.

```
MVI    A, A3H
OUT   X1
```

; Write display RAM from 0000 location auto-increment mode.

;

1	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---

= 90H

```
LXI   H, RD-CD
MVI   C, 05H ; 5 bytes to be displayed
MVI   A, 90H
OUT   X1
```

```
RDISP:      MOV   A, M
            OUT   Y0
```

```

INX    H
DCR    C
JNZ    RDISP
RET
RD-CD           DB     AFH, 86H, 88H, C0H, 91H
                  ISR – RST 6.5

```

; 15 seconds have elapsed after the first key is pressed. Thus, the key pressed must be reset

; to zero and READY message to be displayed, interrupt to be enabled

;

```

MVI    D, 00H ; KPRSD variable is in register D
CALL   RD-DISP
EI
RET

```

ISR – RST 7.5

; Same action as ISR—RST 6.5

Other subroutines remain the same with minor modifications mentioned above. The modified main program instructions are as follows.

;

; Main Program

;

; Initialize Intel 8279.

;

; Set the Keyboard/Display mode.

```

MVI    A, 00H
OUT   X1

```

;

; Program Clock.

; Assume that the external clock frequency is 2 MHz. To obtain 100 kHz internal

; frequency, PPPPP = 10100. Thus, Program Clock command = 0 0 1 1 0 1 0 0 = 34H.

;

```

MVI    A, 34H
OUT   X1

```

;

; Initialize Counter 0.

```

;

    MVI    A, 34H ; Control word
    OUT    0FH
    MVI    A, 84H ; Lower byte of count
    OUT    0CH
    MVI    A, 1EH ; Higher byte of count
    OUT    0CH

;

; Initialize Counter 1.

;

    MVI    A, 50H ; Control word
    OUT    0FH
    MVI    A, 0FH ; Lower byte of count only
    OUT    0DH

;

; Initialize Counter 2.

;

    MVI    A, 90H ; Control word
    OUT    0FH
    MVI    A, 1EH ; Lower byte of count only
    OUT    0EH

;

; Display rEADY

;

    CALL   RD-DISP

;

; Program

;

```

—
—
—

The 8086-based design

Figure 7.60 shows the 8086 interfaced to the 8279 and the 8253/8254. The 8279, in turn, is interfaced to eight seven-segment LEDs and an 8 × 8 keyboard. The IRQ pin of the 8279 is connected to IR0 of octal buffer.

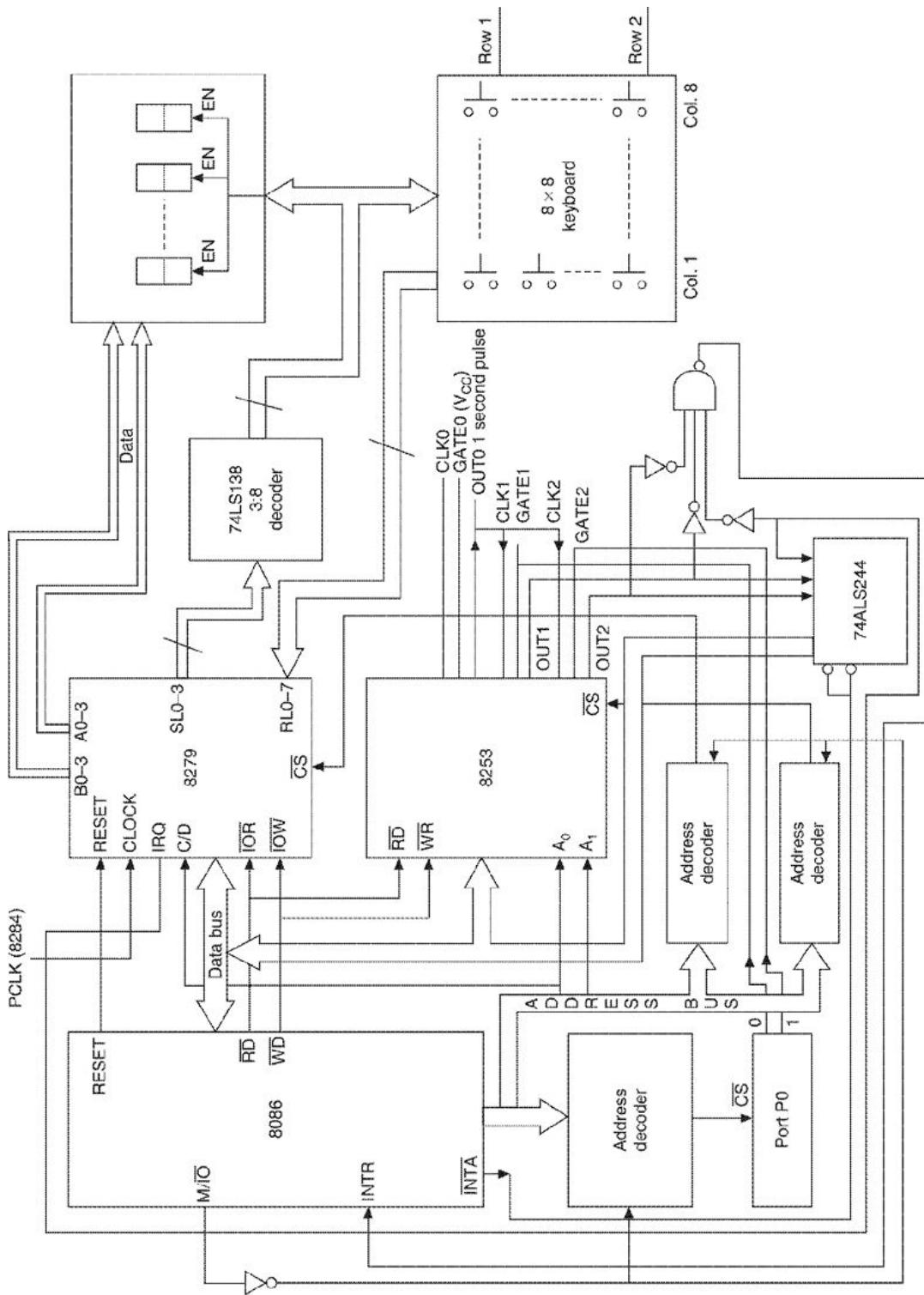


Figure 7.60 Schematic circuit using the 8086 (with the 8253/8254) for automatic ticket counter at metro railway.

We require two types of reset—a 15 second reset and a 30 second reset. Let us allocate the three timer/counters of the 8253 in the following manner:

Counter 0 — For production of 1 second clock pulse which will be input to Counter 1 and Counter 2.

Counter 1 — For 15 second reset.

Counter 2 — For 30 second reset.

Let us assume the following port addresses for the 8253/8254.

Counter 0 — 0CH

Counter 1 — 0DH

Counter 2 — 0EH

Control Word — 0FH

The outputs of Counter 1 and Counter 2, i.e. OUT1 and OUT2 have been connected to IR1 and IR2 respectively. The ISR for IR1 and IR2 will perform the necessary tasks of resetting.

The octal buffer has been used to expand the INTR interrupt to three interrupts. When the interrupt is acknowledged through $\overline{\text{INTA}}$, the interrupt type is put on the data bus and the corresponding ISR is invoked (Figure 7.61). The interrupt types and their locations in the vector table are as follows:

<i>Interrupt</i>	<i>Source type</i>	<i>ISR address</i>	<i>ISR name</i>
IR0	31H (IRQ from Intel 8279)	00C4H	KPIRS
IR1	32H (OUT 1 from Counter 1)	00C8H	CNTIIRS1
IR2	34H (OUT 1 from Counter 2)	00D0H	CNTIIRS2

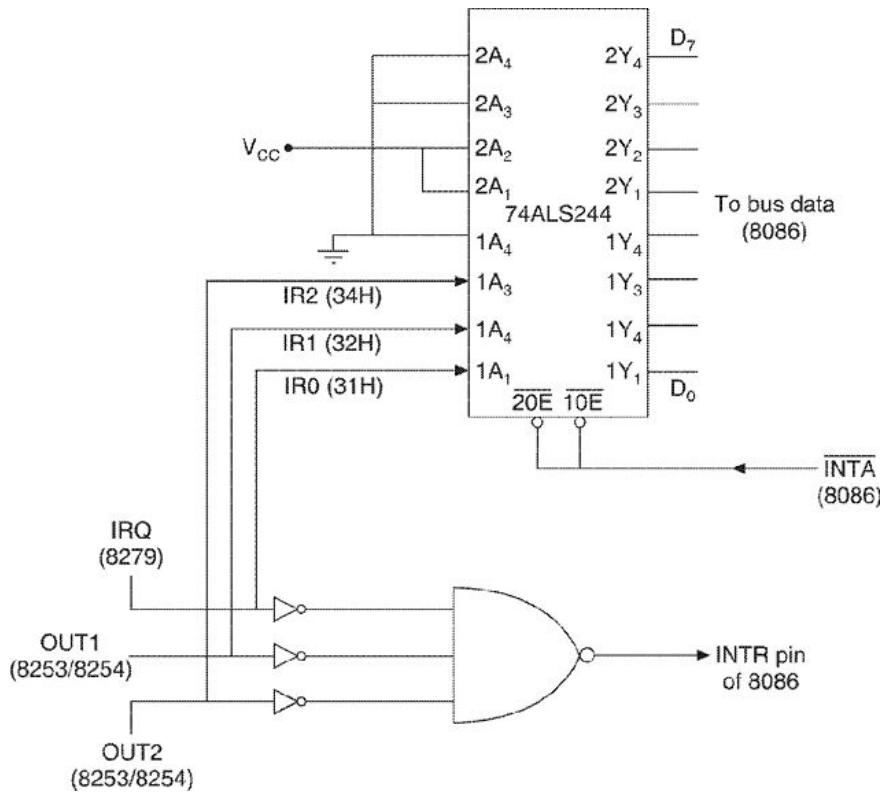


Figure 7.61 Interrupt expansion using octal buffer 74ALS244.

Counter 0: Let us assume that the input clock frequency of the 8086 is 5 MHz. The PCLK of the 8284 clock generator will have a frequency of 2.5 MHz. It is divided by 256 using two 7493s to get 9.765 kHz clock (Figure 7.62). The time period of the clock is 0.1024 millisecond. This is fed as clock to Counter 0 of the 8253 to derive 1 second clock pulse at OUT0.

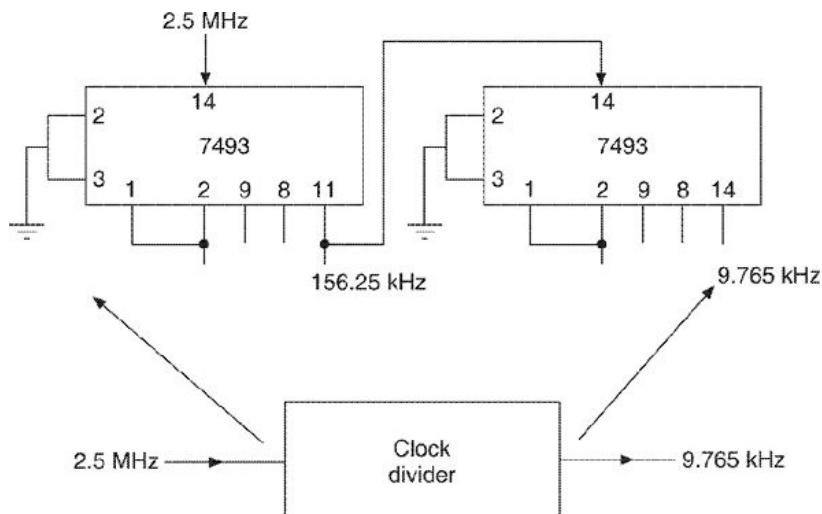


Figure 7.62 Generation of clock for counter.

The counter is used to generate a 1 second pulse. The mode is therefore 2, i.e. the rate generator. It is selected as binary counter and

both the lower-order and higher-order bytes of the count value are loaded into the counter. For 0.1024 millisecond clock, the count value will be 9765. It will be represented as 2625H.

The control word will be:

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

$$= 34H$$

Since the counter is working continuously, V_{CC} is connected to GATE0.

Counter 1: Counter 1 is used to generate a 15 second pulse to reset the system, if no key has been pressed in the last 15 seconds. The OUT1 output line has been connected to IR1, and the ISR for IR1 (i.e. CNTIRS1) will perform the required functions. However, if a key has been pressed for the Destination Station Code, the interrupt should not then occur, i.e. the interrupt should be disabled by the ISR for IR0 (i.e. KPIRS) when the destination key press is detected. The mode will be 0, i.e. interrupt on the terminal count. It is a binary counter and since the count is only 15, only the least significant byte needs to be loaded.

Thus control word will be

0	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

$$= 50H$$

The counter will be started as soon as the first key press is detected. Thus, the high level at GATE1 needs to be programmed. It has been achieved through a Port P0 in Figure 7.60. The P0.0 and P0.1 lines of P0 are connected to GATE1 and GATE2 respectively.

Thus when 01 is written to Port P0 using

```
MOV AL, 01H
OUT P0, AL
```

the GATE1 gets high-level signal and Counter 1 gets activated.

Counter 2: Similar to Counter 1, Counter 2 is used to generate a 30 second pulse to reset the system when the Station Numbers and the Fare have been displayed. The OUT2 output line has been connected to IR2, and the ISR for IR2 (i.e. CNTIRS2) will perform the required functions. The mode will be 0, i.e. interrupt on the terminal count. It is a binary counter and since the count is only 30, only the least significant byte needs to be loaded.

Thus the control word will be

1	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---

= 90H

The counter will be started as soon as the Station Numbers and the Fare have been displayed. Like Counter 1, the activation of Counter 2 can also be programmed using Port P0. Since P0.1 line is connected to GATE2, the following instruction will send a high-level signal at GATE2 thus activating Counter 2.

```
MOV AL, 02H
OUT P0, AL
```

Display: Let us now work out the display of rEADY on the seven-segment LED displays. A total of five seven-segment LEDs will be used in the following manner

Letter	Segments	Display code byte		
r	e, g	1	0	1
E	a, d, e, f, g	1	0	0
A	a, b, c, e f, g	1	0	0
D	a, b, c, d, e, f	1	1	0
Y	b, c, d, f, g	1	0	1

The above display code bytes need to be loaded in the display RAM of the 8279 in sequence. Let us assume that the five display code bytes are stored in consecutive memory locations as RD–CD and a procedure RD–DISP will display the rEADY message when invoked.

RD–DISP PROC NEAR

; Let us erase, if any information was displayed previously using the display write

; inhibit/blanking command.

1	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

; = A3H

; Both nibbles blanked.

```
MOV AL, A3H
OUT X1, AL
```

; Write display RAM from 0000 location auto-increment mode.
Command =

;

1	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---

```

; = 90H
LEA    BX, RD-CD
MOV    CX, 05H ; 5 bytes to be displayed
MOV    AL, 90H
OUT    X1, AL
RDISP :           MOV    AL, (BX)
                  OUT   Y0, AL
                  INC   BX
                  LOOP  RDISP
                  RET
RD-DISP      ENDP

;

; ISR for IR1
;

CNTIRS1      PROC   NEAR
;

; 15 seconds have elapsed after the first key pressed. Thus, KPRSD
must be reset to zero
; and rEADY message to be displayed, interrupt to be enabled
;
MOV    KPRSD, 00H
CALL   RD-DISP
EI
IRET
CNTIRS1      ENDP

;

; ISR for IR2
;

CNTIRS2      PROC   NEAR
;

; Same action as ISR for IR1 (i.e. CNTIRS1)

The other procedures remain the same except for some minor
modifications mentioned above. The modified Main Program is as
follows:
;

; Main Program
;

```

```

DATA SEGMENT
    ST-CD      DW      -, -, -, -, -, -, —; Display
    codes for
                                ; stations
    DIS-CD      DB      8 DUP (0) ; Codes for
    display
    KPRSD      DB      0
    X1         EQU     —
    X0         EQU     —
    Y0         EQU     —
    START-STN   DSB     1
    DEST-STN   DSB     1
    TEMP        DB      0
    COL         DB      0
    ROW         DB      0
    BCODE1      DB      0
    BCODE2      DB      0
    RD-CD      DB      AFH, 86H, 88H, C0H, 91H
DATA ENDS
S-STACK SEGMENT
    DW      100 DUP (0)
T-STACK      LABEL WORD
S-STACK ENDS
;
CODE SEGMENT
ASSUME CS: CODE, DS: DATA, SS: S-STACK
;
; Initialize the segment registers.
;
START:      MOV     AX, DATA
            MOV     DS, AX
            MOV     AX, S-STACK
            MOV     SS, AX
            MOV     SP, OFFSET T-STACK
;
; Initialize Intel 8279.
;
            MOV     AL, 00H
            OUT     X1, AL

```

; Assume Intel 8086 frequency is 5 MHz. Thus PCLK frequency is 2.5 MHz.

; It is divided by 25 using Program Clock Command to get 100 kHz.
Thus—

; Program Clock Command = 0 0 1 1 1 0 0 1 = 39H.

;

MOV AL, 39H

OUT X1, AL

;

; Insert the address of interrupt service routines KPIRS, CNTIRS1 and CNTIRS2 in

; interrupt vector table.

;

MOV AX, 0000H

MOV ES, AX

; Load the address of ISR KPIRS in interrupt vector table.

MOV WORD PTR ES: 00C4H, OFFSET KPIRS

MOV WORD PTR ES: 00C6H, SEG KPIRS

; Load the address of ISR CNTIRS1 in interrupt vector table.

MOV WORD PTR ES: 00C8H, OFFSET
CNTIRS1

MOV WORD PTR ES: 00CAH, SEG CNTIRS1

; Load the address of ISR CNTIRS2 in interrupt vector table.

MOV WORD PTR ES: 00D0H, OFFSET
CNTIRS2 MOV WORD PTR ES: 00D2H, SEG
CNTIRS2

;

; Initialize Counter 0.

;

MOV AL, 34H ; Control word

OUT 0FH, AL

MOV AL, 25H ; Lower byte of count

OUT 0CH, AL

MOV AL, 26H ; Higher byte of count

OUT 0CH, AL

;

; Initialize Counter 1.

;

MOV AL, 50H ; Control word

```

        OUT    0FH, AL
        MOV    AL, 0FH ; Lower byte of count only
        OUT    0DH, AL
;
; Initialize Counter 2.
;
        MOV    AL, 90H ; Control word
        OUT    0FH, AL
        MOV    AL, 1EH ; Lower byte of count only
        OUT    0EH, AL
;
; Display rEADY.
;
        CALL   RD-DISP
;
; Program
;
        —
        —
        —
CODE ENDS

```

EXERCISES

1. Redesign the system of Example 7.5 to display the following messages
ENTER STN.CD
WAIT
ERROR
rEADY
at appropriate stages.
2. Redesign the system of Example 7.5 to include 200 stations.

7.8 DIGITAL-TO-ANALOG CONVERTER

Digital-to-analog converters (DACs) translate digital information into equivalent analog voltage or current. The DAC works on a very simple principle of decoder resistance network. The main constituents of DAC are the decoder network, an analog switch for each digital input, a buffer amplifier and the necessary control logic as shown in Figure 7.63. The

digital interface provides the storage of input data word and the control of the analog switch position. The analog switches under the control of digital interface either connect the reference source to the decoder network input terminal or ground the terminal.

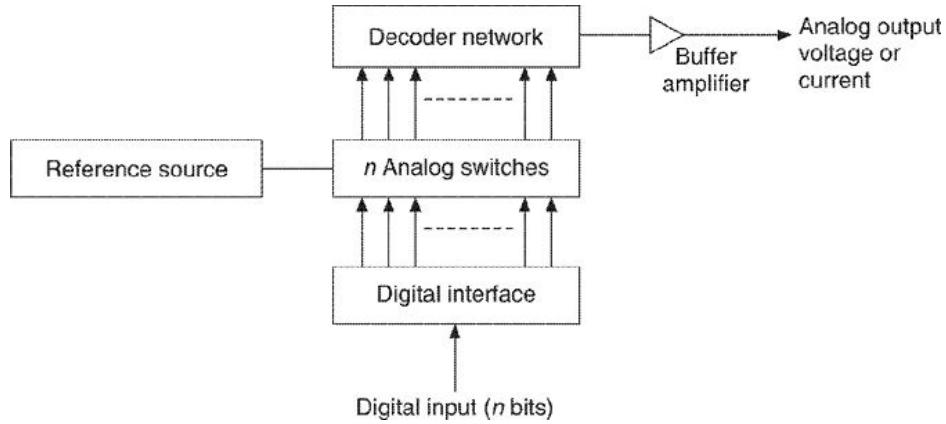


Figure 7.63 Digital-to-analog converter.

The buffer amplifier provides impedance buffering and current driving capability. It is generally required because the output impedance of the decoder network is relatively high. Figure 7.64 shows the pin diagram of a 12-bit DAC.

It is quite easy to interface a digital-to-analog converter to a microprocessor. If an n -bit microprocessor is to be interfaced to an m -bit DAC, the following three possibilities may arise.

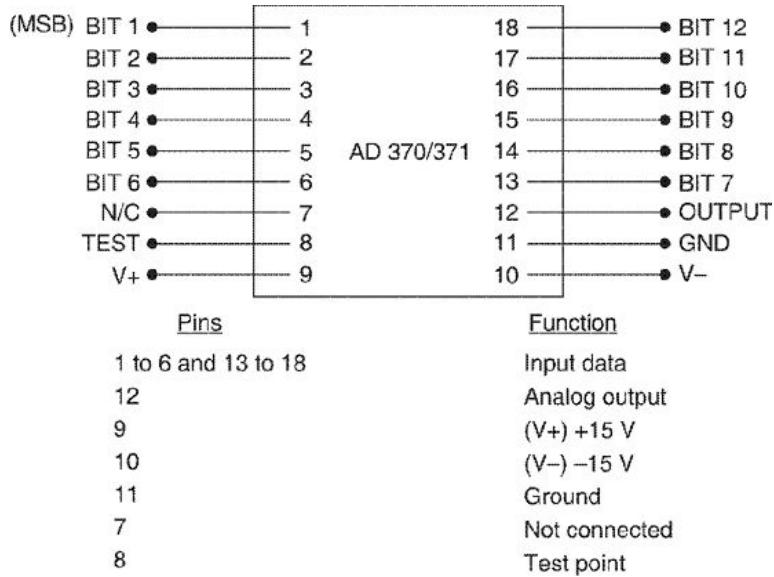


Figure 7.64 DAC AD 370/371 pin diagram.

Case (i) $n = m$: The output of the microprocessor can be directly connected to DAC through a latch as shown in Figure 7.65. The microprocessor outputs the 8-bit digital value to the specified latch

which is connected to the DAC. The software interface consists of instructions to load the digital data to the latch.

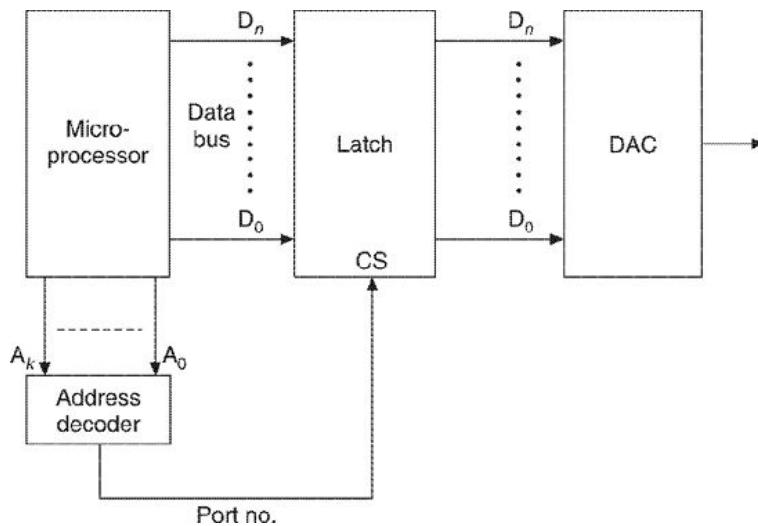


Figure 7.65 Microprocessor interface to DAC ($n = m$).

Case (ii) $n > m$: This case is abnormal. However, if one is forced to connect a higher-bit (n) microprocessor to a lower-bit (m) DAC, only then, the lower m bits of the microprocessor data bus are connected to the DAC latch.

For DAC, the microprocessor behaves like an m -bit microprocessor (Figure 7.66). The programmer must ensure that the data is represented only in m bits, otherwise the higher-order bits may be lost, resulting in loss in accuracy. The software consists of loading data into the latch.

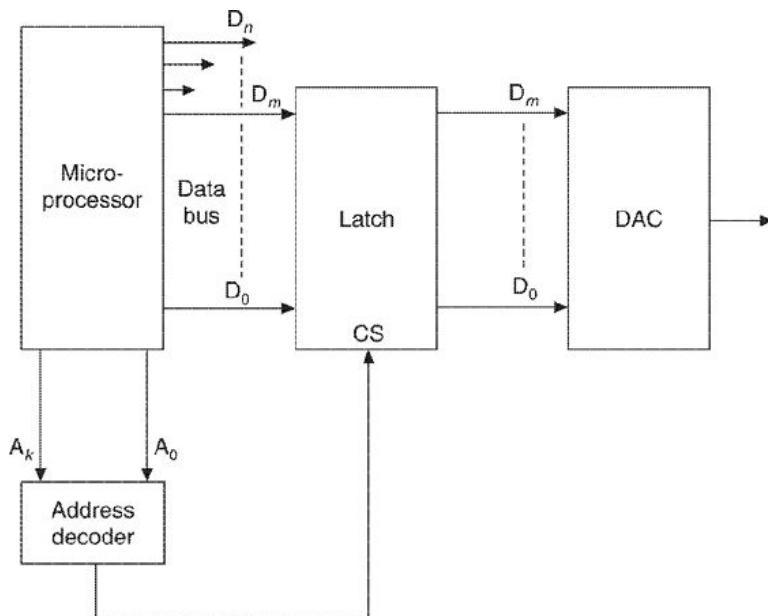


Figure 7.66 Microprocessor interface to DAC ($n > m$).

Case (iii) $n < m$: Often it happens that 16-bit calculations are required to be performed on an 8-bit microprocessor. This is achieved by using double precision. If this result is to be output on a 16-bit DAC, then the 16-bit DAC will have to be interfaced to an 8-bit microprocessor. Since DAC works continuously without any start/stop pulse, the two bytes should be loaded to DAC at the same instant. If the two bytes of the 16-bit data are loaded one by one, then the analog value at the output will not represent the 16-bit digital value. Thus, interfacing a lower bit (n) processor to a higher bit (m) DAC is quite tricky.

The interface block diagram is shown in Figure 7.67. First the lower-order n -bit data are stored in latch 1. The higher-order $(m - n)$ bits are then stored in latch 2. The m -bit data are then stored simultaneously in latch 3 and latch 4 using the same pulse. Latch 3 and latch 4 are connected to DAC directly. The software involves storing the data in various latches. The latch 3 and latch 4 are selected simultaneously and thus the DAC gets all the m bits of data at the same instant.

Some DACs output current while others output voltage as analog values. Often, it is required to convert voltage to current and vice versa. This can be achieved through an operational amplifier circuit. As operational amplifiers are out of the scope of this book, we shall not discuss this aspect here. However the interested readers may refer (Connelly, 1975).

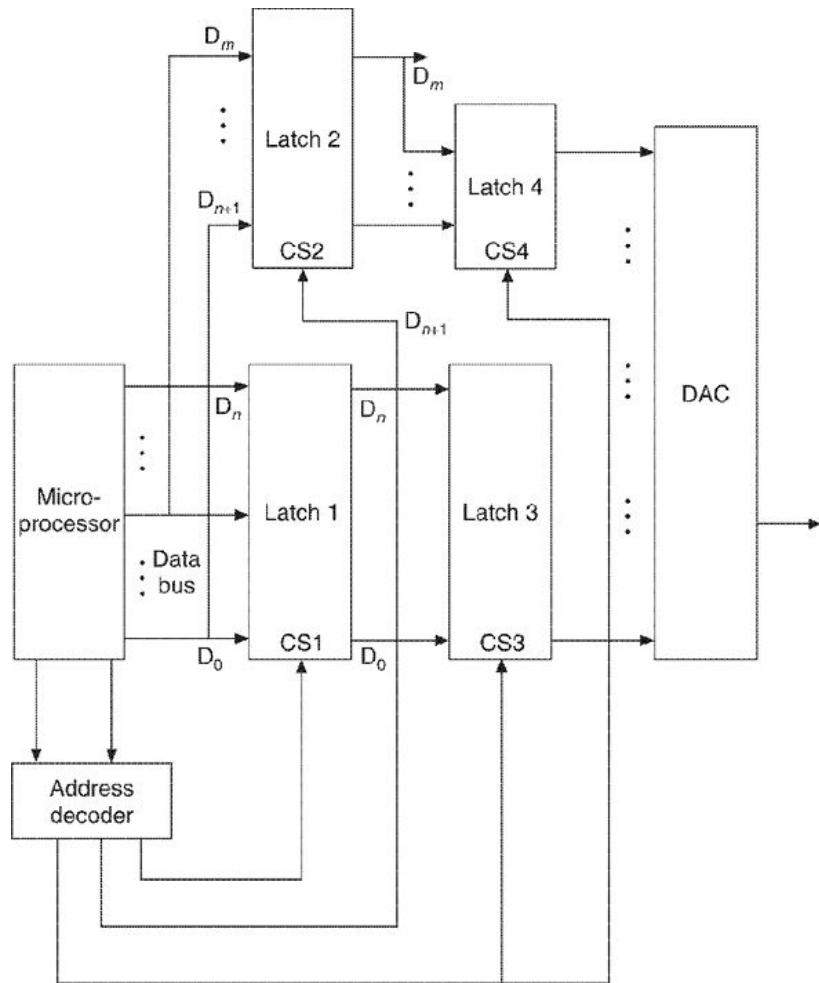


Figure 7.67 Microprocessor interface to DAC ($n < m$).

EXAMPLE 7.6

In an eight-floor air-conditioned office, the air is supplied through eight air outlets (one for each floor) each having a control valve to regulate the air flow. The air flow at each floor is set through a panel which contains one knob for control valve to regulate the air flow speed from minimum to maximum in eight steps. Thus, when a particular knob is at zero (min), the valve is fully closed and the air flow is zero. On the other hand, when the knob is at seven (max), the valve is fully open and the air flow is maximum.

The air temperature is kept constant depending on the ambient temperature. In summer, the air will be cool, whereas in winter it will be hot.

Our task is to develop a microprocessor-based system to regulate the air flow in each floor based on the settings.

Solution:

Let us begin by understanding the working of the electronic control valve. The air flow regulation using a control valve is shown in Figure 7.68.

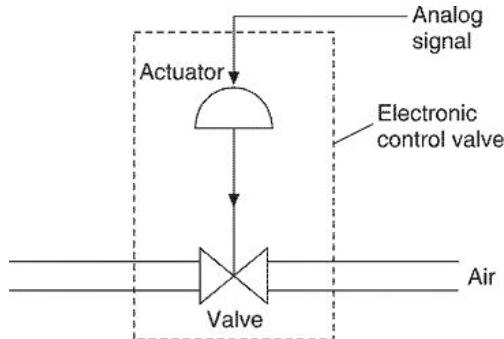


Figure 7.68 Air flow regulation.

The actuator in the electronic control valve receives the analog signal corresponding to the desired air flow and it moves the flaps in the valve to regulate the air flow. The DAC can be used to convert the digital value of the desired flow to the equivalent analog signal which can be accepted by the electronic control valve.

The schematic circuit using the 8085 is shown in Figure 7.69. The circuit using the 8086 microprocessor is shown in Figure 7.70.

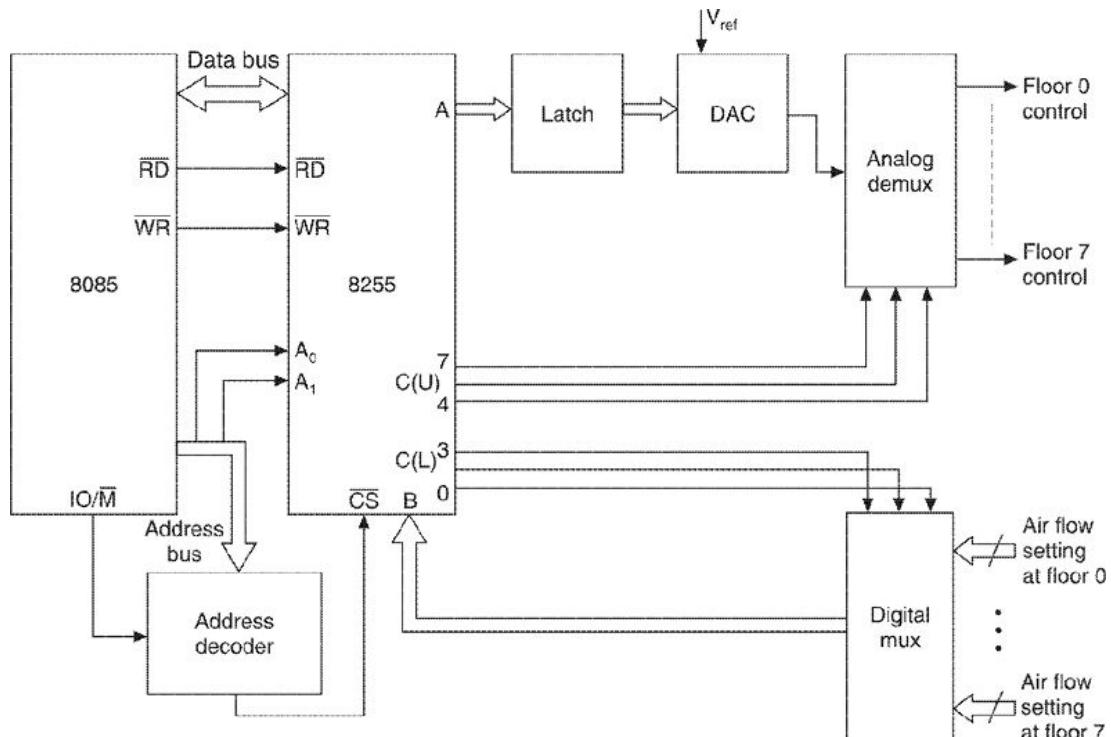


Figure 7.69 Schematic circuit using the 8085 for air-conditioning control.

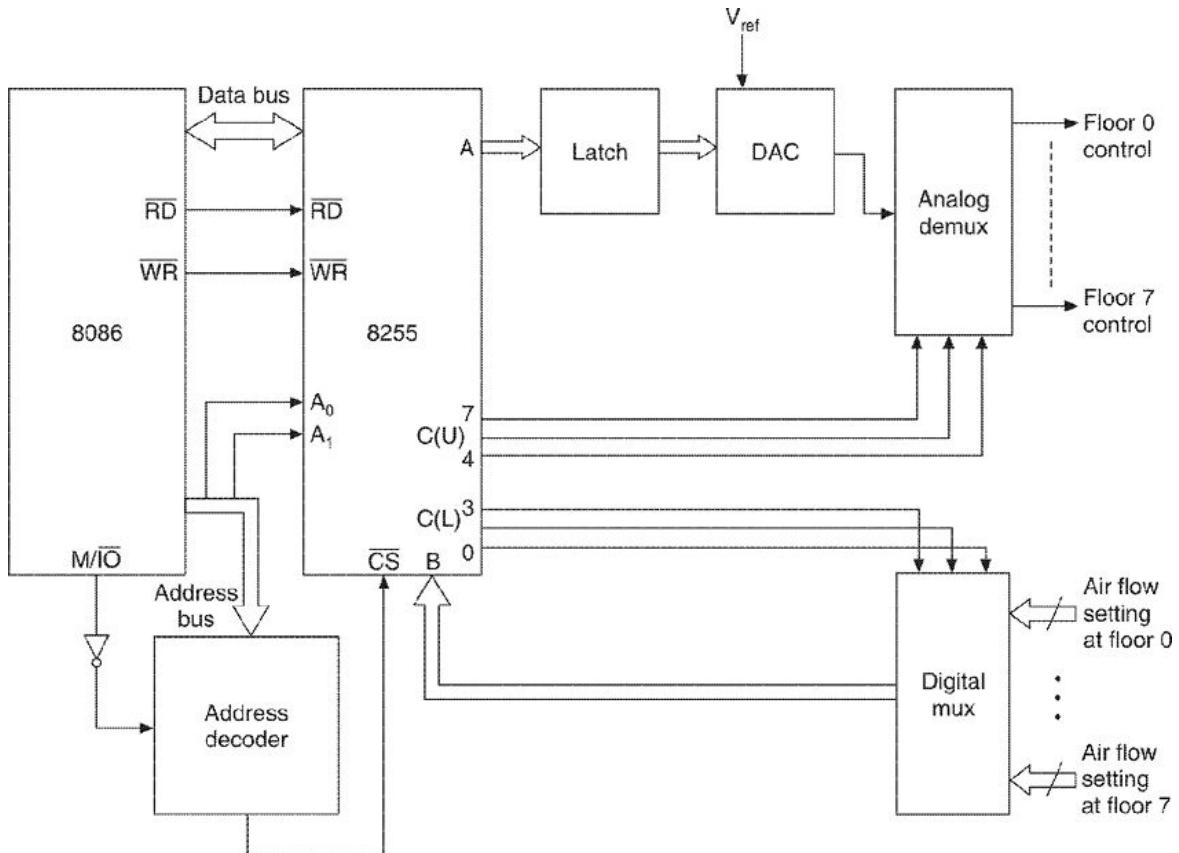


Figure 7.70 Schematic circuit using the 8086 for air-conditioning control.

From each knob, eight wires emanate, one for each setting. Only one of the eight wires will have a high-level signal while the others will have low level signals. These eight sets (for eight floors) of eight wires are fed to a digital multiplexer. The settings for different floors are read by the microprocessor in sequence and control action (if any) is determined. The control valve position in digital form is converted to analog voltage by DAC. The analog voltage is then demultiplexed and sent to the concerned electronic control valve.

The 8255 PPI has been used to interface DAC, analog demultiplexer, and digital multiplexer with the following port assignments

- Port A : Output Control valve position to DAC through latch
- Port B : Input 8-bit floor settings from digital multiplexer.
- Port C(U) : Output Address lines for selection of control valve in analog demultiplexer
- Port C(L) : Output Address lines for selection of air flow settings at different floors through digital multiplexer.

Let us assume that the 8255 address is 30H. Thus different ports will have the following addresses:

- Port A : 30H

Port B : 31H
 Port C : 32H
 Control Word : 33H

The mode will be 0, i.e. the basic input-output. The 8255 initialization control word will be

1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

$$= 82H$$

Thus the value 82H will be moved to control word register for initialization.

The eight settings of air flow can be visualized as

7	6	5	4	3	2	1	0

One bit corresponds to each air flow setting. Thus, if bit 3 = 1, the air flow control valve is at the third position.

Let us assume that the control valve position for each air flow setting has been pre-calculated and stored in a look-up table in memory starting from location CVSET.

CVSET	control valve position for 0 setting
CVSET+1	control valve position for 1 setting
⋮	⋮
CVSET+7	control valve position for 7 setting

Thus the microprocessor must read an air flow setting, determine the control valve position through the look-up table and send it to the control valve through DAC.

Is it all?

No. Different situations may also occur.

Most of the times, there may not be any change in the air flow setting. In such cases, no change in the control action will be desired. To facilitate this, let us assume that the processor maintains another look-up table starting from location CVPOS.

CVPOS	control valve position for floor 0
CVPOS+1	control valve position for floor 1
⋮	⋮
CVPOS+7	control valve position for floor 7

Now we shall develop programs using the 8085 and 8086 microprocessors.

Using the 8085 microprocessor

MVI A, 82H
OUT 33H

; Set floor counter, register C is used for this purpose

INITIATE: MVI C, 00H

; Read the air-flow setting.

AIR-FLOW: MOV A, C
OUT 32H
IN 31H

; ACC contains the air-flow settings. Determine the control valve position.

MVI D, 00H

; Rotate right ACC through carry to find setting.

START: RRC
JC NEXT
INR D

; Compare with 7 to find if the rotation is complete.

MOV E, A ; E is temporary register
MOV A, D
CPI 07
JZ ERROR
MOV A, E
JMP START

; D contains the knob setting for air flow. Now find the control valve setting.

NEXT: LXI H, CVSET
MOV A, L
ADD D
MOV L, A
MOV A, M ; (A) = Control valve position
MOV E, A ; E—Temporary storage

; Determine the present position.

LXI H, CVPOS
MOV A, L
ADD C ; (C) = Floor no.
MOV L, A
MOV A, M ; (A) = Present position

; Find if both are same.

```
SUB E  
JNZ ACTION
```

; Both are same. No action. Repeat action for other floors.

NEXT-FLOOR: INR C ; (C) = Next floor no.
MOV A, C
CPI 08H
JNZ AIR-FLOW

; One cycle completed. Start the next cycle.

```
JZ INITIATE
```

; Output the control valve position.

ACTION: MOV A, C
RLC
RLC
RLC
RLC

; Now, floor no. is in upper nibble. Select the control valve through analog DEMUX.

```
OUT 32H
```

; Send the valve position.

```
MOV A, E  
OUT 30H
```

; Change in the control valve position. Repeat for other floors.

```
LXI H, CVPOS  
MOV A, L  
ADD C  
MOV L, A  
MOV A, E  
MOV M, A  
JMP NEXT-FLOOR  
CVSET DSB 8  
CVPOS DSB 8  
ERROR: HLT
```

Using the 8086 microprocessor

;
; Intel 8086 program for air-conditioning control using ADC and DAC

```

;

        DATA SEGMENT
            CVSET      DB      -, -, -, -
            CVPOS      DSB     8
            TEMP       DB      0
        DATA ENDS
        CODE SEGMENT
ASSUME CS: CODE, DS: DATA
        MOV AX, DATA
        MOV DS, AX
        MOV AL, 82H
        OUT 33H, AL
NCYCLE:      MOV CX, 0000H
;
; Read the air-flow setting.
;
AIRFLO:      MOV AX, CX
        OUT 32H, AL
        IN  AL, 31H
;
; (AL) = Air-flow setting. Determine the control valve position.
;
        MOV DX, 0000H
; Rotate right AL through carry.
START: RCR AL, 01
        JC NEXT
        INC DX
;
; Compare with 07H to find if the rotation is complete.
        CMP DX, 0007H
        JZ ERROR
        JMP START
;
; (DX) = Knob setting for air flow. Find the control valve setting.
NEXT:      LEA BX, CVSET
        ADD BX, DX
        MOV AL, (BX)
;
; (AL) = Control valve position.
        MOV TEMP, AL
;
; Determine the present position.

```

```

        LEA    BX, CVPOS
        ADD    BX, CX ; (CX) = Floor No.
        MOV    AL, (BX)
;
; Find if the Control Valve Setting = Present Position?
;
        SUB    AL, TEMP
        JNZ    ACTN
;
; Both are same. No action. Repeat for other floors.
;
NXFLOOR:           INC    CX
                  CMP    CX, 08
                  JNZ    AIRFLO
;
; One cycle completed. Start the next cycle.
;
        JMP    NCYCLE
;
; Output the control valve position.
;
ACTN:             MOV    AX, CX
                  ROL    AL, 01
                  ROL    AL, 01
                  ROL    AL, 01
                  ROL    AL, 01
;
; Now floor no. is in the upper nibble of AL. Select the control valve
; through analog DEMUX.
        OUT    32H, AL
;
; Send the valve position.
        MOV    AL, TEMP
        OUT    30H, AL
;
; Change the control valve position. Repeat for other floors.
        LEA    BX, CVPOS
        ADD    BX, CX
        MOV    (BX), TEMP
        JMP    NXFLOOR
        HLT
CODE ENDS

```

EXERCISE

1. The eight floor office in Example 7.6 is divided into a hall and a corridor in each floor. Both hall and corridor have separate air outlets (now, total 16) and separate knobs to set the air flow. Redesign the system to include the corridor air-flow control as well.

7.9 ANALOG-TO-DIGITAL CONVERTER

The analog-to-digital converter (ADC) takes an analog signal as input and presents an equivalent digital signal as output. The basic techniques include—ramp, integrating ramp, successive approximation, etc. Out of these, the successive approximation technique is quite prevalent. These techniques are not discussed here as they are outside the scope of this book. You may refer (Connelly, 1975; Leventual, 1978) to get the details on these techniques.

The configuration of an ADC chip is shown in Figure 7.71. The whole operation is carried out by the following signals.

- Start Convert
- End of Conversion
- Output Enable

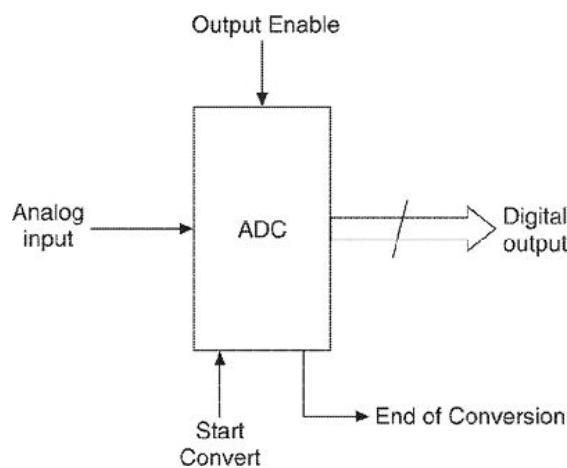


Figure 7.71 ADC chip configuration.

The control logic sends the Start Convert signal when the analog signal is stable on the analog input line. The ADC converts the analog signal to digital and sends the End of Conversion signal on completion of conversion. To read the output from digital output lines, the control logic

should send the Output Enable signal to ADC, which will then enable the lines of the digital output. In many ADCs the digital output is sent to digital output lines by ADC along with the End of Conversion signal. Thus the Output Enable is not used in such cases.

As discussed above, the analog-to-digital converter (ADC) deals with the following signals.

Input voltage	Input analog voltage. Input to ADC.
Start Convert	Command to start conversion. Input to ADC.
End of Conversion	Information that conversion is complete and the output can be read. Output signal from ADC.
Digital output	Digital value of analog input derived from conversion. Output from ADC.

A microprocessor can be interfaced to ADC in any one of the following three modes.

7.9.1 Asynchronous Mode

Figure 7.72 shows the microprocessor interface with ADC in asynchronous mode. The microprocessor starts the conversion process by sending the Start Convert signal to the ADC. The ADC takes the analog input and converts it to digital. When the conversion is complete, the ADC outputs the End of Conversion signal to the microprocessor. On receiving this signal,

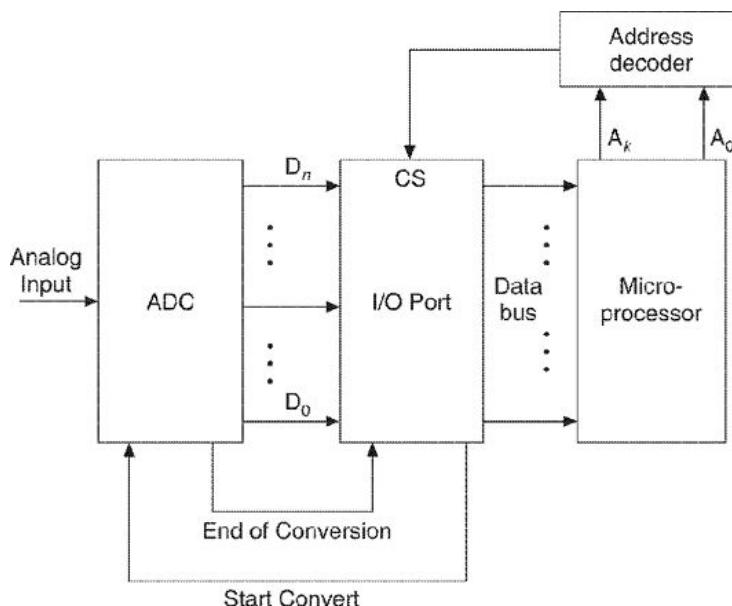


Figure 7.72 Microprocessor interface to ADC(asynchronous mode).

the microprocessor reads the data present at the output of ADC. The software flowchart for interface is presented in Figure 7.73. The

microprocessor continuously checks for the End of Conversion signal after the Start Convert signal is output.

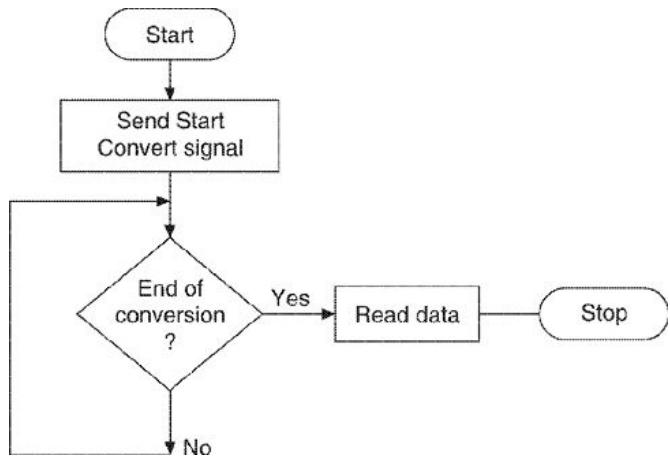


Figure 7.73 Asynchronous mode ADC–microprocessor interface software flowchart.

7.9.2 Synchronous Mode

It is evident that in the asynchronous mode, the microprocessor wastes time by waiting for the End of Conversion signal. The conversion time for an ADC is up to a few milliseconds. Thus, for every conversion, this much time of the microprocessor will be wasted.

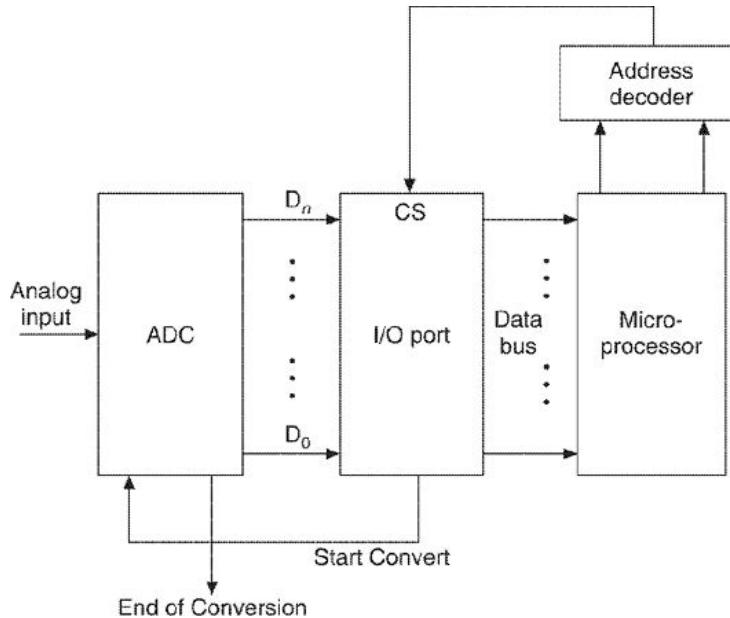


Figure 7.74 Microprocessor interface to ADC (synchronous mode).

Figure 7.74 shows the microprocessor interface with ADC in the synchronous mode. The microprocessor issues the Start Convert signal to initiate the conversion process. Since the conversion time is known, the microprocessor, then, executes certain instructions so that the execution time of the instructions is greater or equal to the conversion time. The

microprocessor will then read the data from ADC directly. The software flowchart for the interface is shown in Figure 7.75.

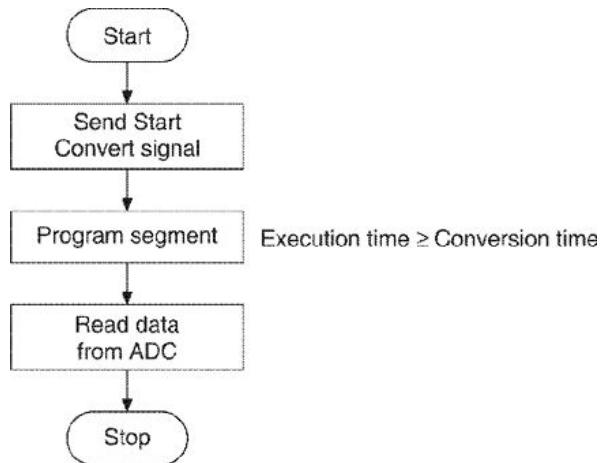


Figure 7.75 Synchronous mode ADC–microprocessor interface software flowchart

The microprocessor utilizes the conversion time in executing a part of the program. This program segment will clearly not use the ADC output. The End of Conversion signal is not used in this interface.

7.9.3 Interrupt Mode

Many users, while accepting the benefits of the synchronous interface, find it inconvenient to write the program segment whose execution time is nearly equal to the conversion time. In a system containing 50 channels, 50 such different program segments will have to be written. The interrupt mode is used to avoid this.

The interrupt mode interface is shown in Figure 7.76 and the software flowchart for the interface is presented in Figure 7.77.

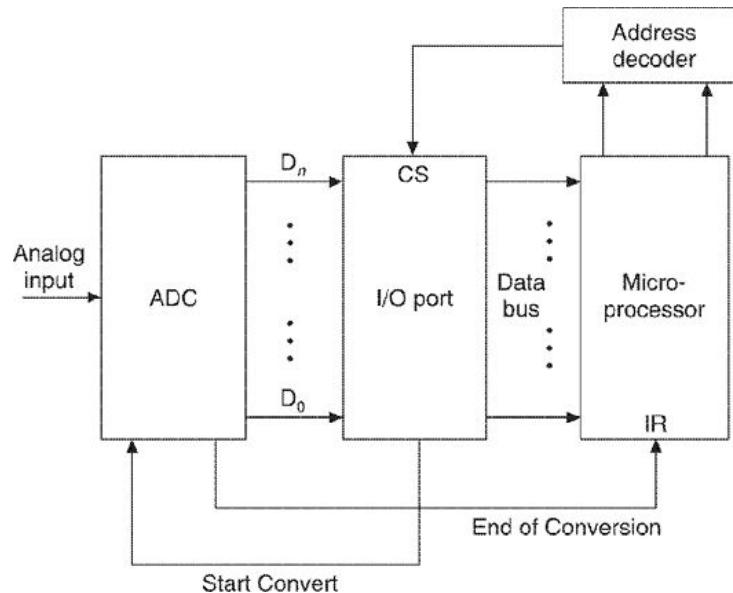


Figure 7.76 Microprocessor interface to ADC (interrupt mode).

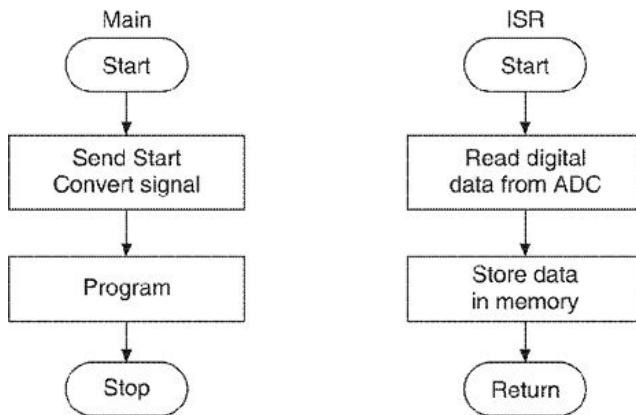


Figure 7.77 Interrupt mode ADC–microprocessor software interface flowchart.

In this mode, the End of Conversion signal is physically connected to one of the interrupt inputs of the microprocessor. The microprocessor initiates the conversion process by sending the Start Convert signal to the ADC. The ADC completes the conversion and sends the End of Conversion signal. Since the End of Conversion signal is connected to one of the interrupt inputs of the microprocessor, the microprocessor is interrupted. The microprocessor suspends its program execution, saves the status and then executes the Interrupt Service Routine to service the interrupt caused. The interrupt service routine, in the present case, contains instructions to read and store the digital data from ADC.

It is clear that the interrupt mode is most advantageous for the interfacing of the microprocessor to the ADC. However, it uses one interrupt input of the microprocessor. Let us take one more example to understand this concept clearly.

EXAMPLE 7.7

Design the air-conditioning control problem of Example 7.6 such that it is made automatic by deciding a temperature setting for the whole building and monitoring the temperature at each floor.

Solution:

In this example, it is not necessary to have an air-flow setting for each floor. The panel and knobs will be replaced by a temperature sensor at each floor.

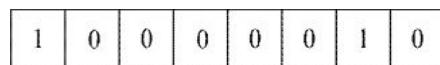
The 8255 may be used, as in the previous example, to interface the input (i.e. the temperature sensors, through an analog multiplexer, S&H and ADC) and the output (i.e. the flow control valves through analog demux, DAC and latch). The port assignments may be as follows:

Port A	Output	Connected to DAC through latch
--------	--------	--------------------------------

Port B	Input	Connected to ADC
Port C	Output	Different lines connected to ADC (Start Convert), S&H (Hold), analog mux (Channel Address) and analog demux (Channel Address).

Let us assume that the 8255 address is 30H, i.e. the address of ports will be A-30H, B-31H, C-32H and the Control Word will be 33H. Only the B port is the input port. Other ports are output ports. The mode is basic input-output

The control word will be



$$= 82H$$

We shall now develop the system using the 8085 and 8086 microprocessors.

Using the 8085

The schematic circuit using the 8085 is shown in Figure 7.78. Following is the software.

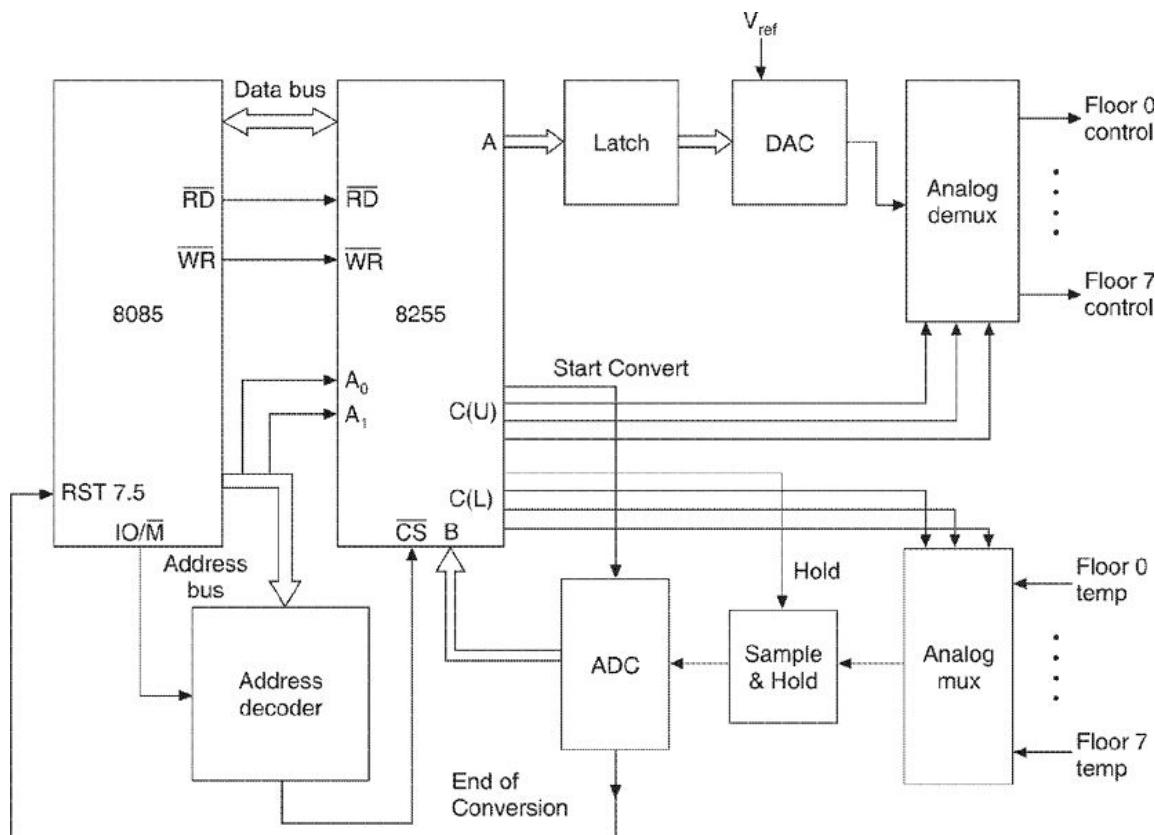


Figure 7.78 Schematic circuit using the 8085 for air-conditioning control (using ADC).

; Intel 8255 initialization

```
MVI A, 82H  
OUT 33H
```

; Set the floor counter register C.

```
START: MVI C, 00H
```

; Read the temperature.

```
TEMP-RD: MOV A, C  
ORI 08H
```

; This will make the third bit = 1 to give hold command to Sample and Hold.

```
OUT 32H
```

; Give the Start Convert signal.

```
MVI A, 80H ; 7th bit = 1  
OUT 32H
```

; RST 7.5 ISR will be invoked when conversion is complete. ISR will store the TEMP

; value in D.reg. and return.

The temperature value read from the sensors is not in the engineering unit of Centigrade or Fahrenheit. It may be converted to the engineering units and displayed if required, by multiplying it with a constant factor which will depend on the type of sensor, ADC etc. Refer (Connelly, 1975; *Peripheral Design Handbook*, Intel Corporation, 1983).

However, if the temperature value is not being displayed, then the value read can be used to take decision. In order to facilitate that the temperature set point is also represented in the same fashion.

Error = Set Point – Actual Temp.

The control valve position can be calculated through the PID algorithm based on error. The set point is stored at the specified location in memory. Let us presume that it is 200FH.

; Calculate the error.

```
LDA 200FH  
SUB D  
CALL PID-CAL
```

; PID subroutine calculates the valve position and stores it in location 2100H and returns

;

; Send valve position to control valve through DAC.

;

```
MOV A, C
RLC
RLC
RLC
RLC
OUT 32H
LDA 2100H
OUT 30H
```

; Repeat for other floors.

```
INR C
MOV A, C
CPI 08H
JNZ TEMP-RD
```

; Cycle completed. Start another cycle.

```
JMP START
HLT
ISR-RST7.5
```

; Read TEMP value.

```
IN 31H
MOV D, A
EI
RET
Subroutine PID-CAL
```

; Subroutine implements the PID algorithm to calculate the valve position from

; error. It may be only proportional, proportional + integral or proportional +

; integral + differential. In the simplest form of proportional control,

; Valve Position = Constant □ Error

; Standard PID routines are available, or they can be developed. Refer (Leventhal, 1978; Krishna Kant, 1987)

Using the 8086

The circuit schematic using the 8086 is shown in Figure 7.79. The End of Conversion signal causes interrupt at INTR, which causes Type 20H, i.e. Type 32 interrupt. The octal buffer 74ALS244 has been used for this purpose.

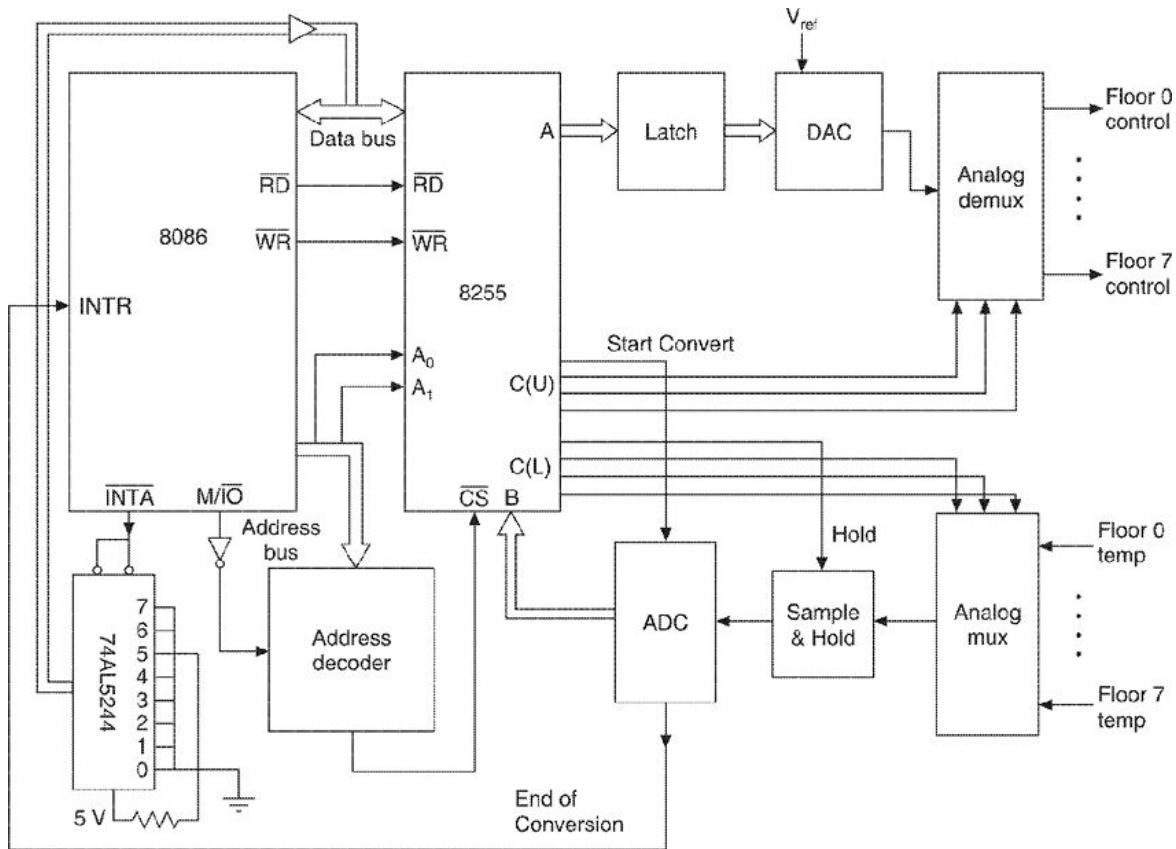


Figure 7.79 Schematic circuit using Intel 8086 for air conditioning control (using ADC).

```

;
; Main Program
; SET-PT = Set Point, VAL-POS = Valve Position, FLRCNT = Floor
Count,
; FLRTMP = Floor Temperature.
;
        DATA SEGMENT
        FLRTMP      DB      0
        FLRCNT     DB      0
        SET-PT      DB      0
        VAL-POS     DB      0
        ERROR       DB      0
        DATA ENDS
        STACK-SEG SEGMENT
        DW 100 DUP (0)
        T-STACK LABEL WORD
        STACK-SEG ENDS
;

```

```

; Make available data segment constants and variables to other
modules.

;

PUBLIC      SET-PT,      VAL-POS,      ERROR,
FLRTMP

;

; Declare that ISR-TEMP is in other module.

PROCEDURES SEGMENT PUBLIC
    EXTRN      ISR-TEMP : FAR
PROCEDURES ENDS
CODE SEGMENT
    ASSUME CS: CODE, DS: DATA, SS: STACK-SEG
;

; Initialize the segment registers.

;

START:      MOV AX, DATA
            MOV DS, AX
            MOV AX, STACK-SEG
            MOV SS, AX
            MOV SP, OFFSET T-STACK

;

; Initialize the 8255.

;

MOV AL, 82H
OUT 33H, AL

;

; Insert the address of ISR-TEMP in the interrupt vector table. Since
the interrupt type is 20H,
; the offset is stored at 0080H, 0081H and the segment address at
0082H, 0083H.

;

MOV AX, 0000H
MOV ES, AX
MOV WORD PTR ES: 0080H, OFFSET ISR-
TEMP
MOV WORD PTR ES: 0082H, SEG ISR-TEMP

; Set Floor Count = 0
NCYCLE:     MOV FLRCNT, 00H

;

; Read the temperature.

```

```

;

TEMP-RD:    MOV   AL, FLRCNT
             OR    AL, 08H

; This makes the 3rd bit = 1 to give hold command.

        OUT   32H, AL

;

; Floor has been selected on Analog Mux and hold command given to
Sample and Hold.

; Give Start Convert signal.

;

MOV   AL, 80H ; 7th bit = 1
OUT   32H, AL

;

;

;

; End of Conversion signal will cause Type 20H interrupt at INTR.

The ISR procedure

; ISR-TEMP will store the floor temperature in memory location
FLRTMP.

; The temperature value read from sensors is not in engineering units
of Centigrade or

; Fahrenheit. It may be converted to engineering units and displayed,
if required, by

; multiplying it with a constant factor which will depend on the type
of sensor, ADC

; etc. Refer (Leventual, 1978; Peripheral Design Handbook, Intel
Corporation, 1983).

; However, if it is not being displayed, as in the present case, then

; the value read can be used to take the control decision. To facilitate
that, temperature

; set point is also represented in the same fashion.

; Error = Set Point – Actual Temperature

; Control valve position can be calculated using the PID algorithm
based on error.

; The Set Point is stored in memory as SET-PT.

;

;

; Calculate the Error.

;

MOV   BL, SET-PT
SUB   BL, FLRTMP
MOV   ERROR, BL

```

```

;

; PID-CAL is PID subroutine which calculates the valve position,
stores it as VAL-POS

; and returns.

;

        CALL    PID-CAL

;

; Send Floor no. to Analog demux to select channel no.

;

        MOV     AL, FLRCNT

; Bring Floor no. in upper nibble.

        MOV     CL, 04
        SAL     AL, CL
        OUT    32H, AL

; Send value position to control value through DAC.

        MOV     AL, VAL-POS
        OUT    30H, AL

;

; Repeat for other floors.

;

        MOV     BL, FLRCNT
        INC     BL
        MOV     FLRCNT, BL
        CMP     BL, 08H
        JNZ     TEMP-RD

;

; Cycle completed. Start another cycle.

;

        JMP     NCYCLE
        NOP
        CODE  ENDS

;

;

; ISR ISR-TEMP program

;

        DATA  SEGMENT    PUBLIC
              EXTRN      FLRTMP
        DATA  ENDS

        PUBLIC      ISR-TEMP

```

```

PROCEDURES SEGMENT PUBLIC
ISR-TEMP PROC FAR
ASSUME CS: PROCEDURES, DS: DATA
;
; Read the temperature value from Port B of the 8255.
;
    IN AL, 31H
    MOV FLRTMP, AL
    EI
    IRET
ISR-TMP ENDP
PROCEDURES ENDS

PID-CAL PROC NEAR
;
;
; It implements the PID algorithm to calculate the valve position from
error.
; The control strategy may be proportional, proportional + integral or
; proportional + integral + differential. In the simplest form of
proportional control
; Valve Position = Constant □ Error
; Standard PID routines are available or can be developed.

```

EXERCISE

- Redesign the system of Example 7.7 to include temperature monitoring and control in hall as well as in corridor in each floor. Now, the total number of temperature sensors and temperature controls will be 16 each. Also, no control will be necessary if the temperature is within the 10% range of the set point. The temperature in each floor and corridor may be displayed in a central location for proper monitoring.

(Hint: You may consider serial mode display of the seven-segment LEDs)

7.10 CRT TERMINAL INTERFACE

The CRT terminals work on RS-232 interface. We need to take only three wires from RS-232C interface of the terminal, namely TXD, RXD

and GND. TXD is used for data transmission from CRT to microprocessor, RXD for receiving data at CRT from the microprocessor and GND for common ground. This is the simple interface which we will be discussing in this section. A study of the complete RS-232C interface is beyond the scope of this book.

The RS-232C interface transmits and receives the binary data serially. One byte is sent enclosed between the start and stop bits. The number of start or stop bits may be 1, 1.5 or 2. Figure 7.80 shows the RS-232C format for one stop and one start bit. Binary 1s and 0s are defined in RS-232C by voltages greater than +3 V and less than -3 V respectively. Thus, TTL signals (0–5 V) are to be converted to RS-232 signals and vice versa. Commonly in CRT terminals, -12 V to -3 V is taken as binary zero and +3 V to +12 V as binary 1.

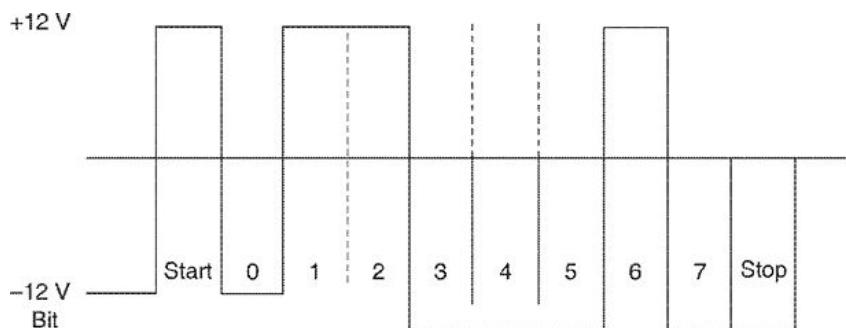


Figure 7.80 RS-232C format.

The microprocessors which have facilities for serial input/output, can be directly connected to the CRT terminal through the signal level converters (Figure 7.81).

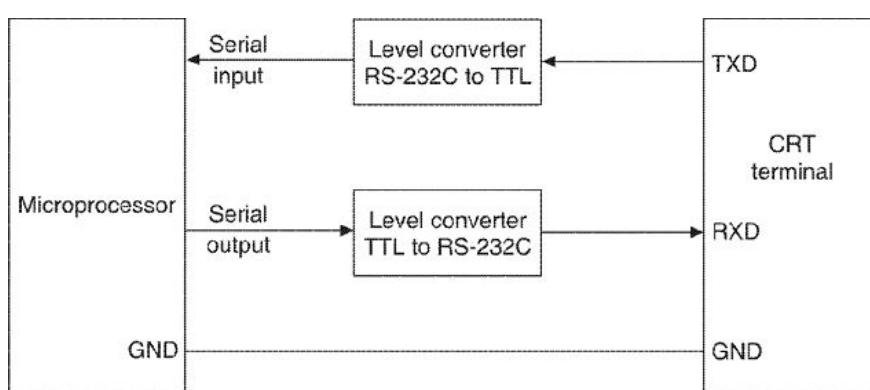


Figure 7.81 Microprocessor interface to CRT terminal (direct).

In case of the microprocessors which have no serial input/output pins, the data on the data bus is converted to serial data through a parallel-to-serial converter. The serial data is made compatible to RS-232C interface by a level converter. In the same way (Figure 7.82), serial data from CRT

is passed through the level converter and serial-to-parallel converter before passing it to the microprocessor through data bus. Normally, specialized IC chips USART(Universal Synchronous Asynchronous Receive Transmit) as shown in Figure 7.83 are used for this purpose.

The software at the microprocessor which controls the interface does the following tasks sequentially.

- Transmission:
 - (a) Send start bit.
 - (b) Send data bits.
 - (c) Send stop bits.
- Receive
 - (a) Check if the start bit has occurred.
 - (b) Receive eight data bits.
 - (c) Check for stop bits.

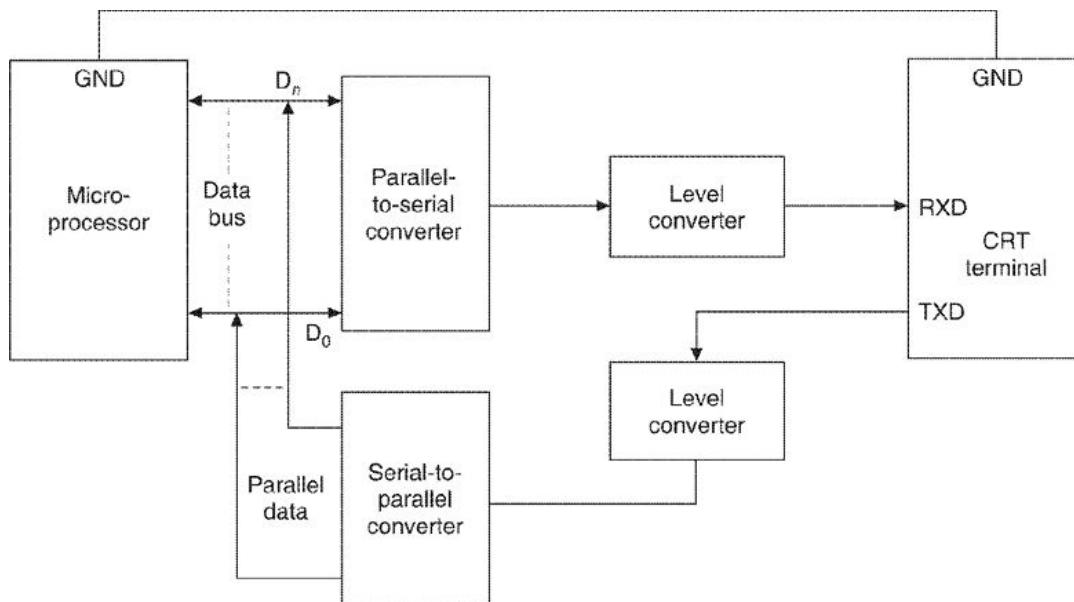


Figure 7.82 Microprocessor interface to CRT terminal (through serial/parallel converter).

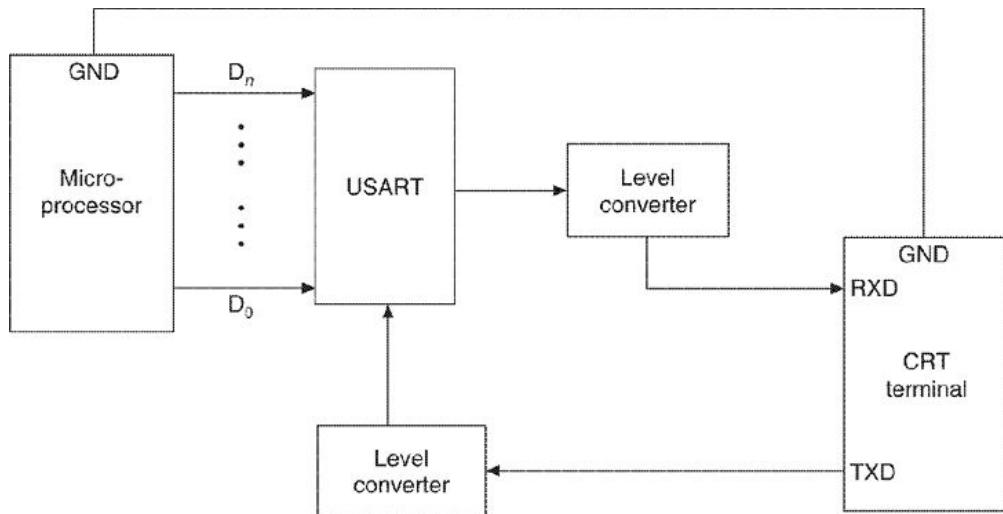


Figure 7.83 Microprocessor interface to CRT terminal (through USART).

The CRT terminal transmits and receives the data at a fixed baud rate (baud is defined as the number of signal events per second. In the present case, it is same as bits per second). A baud rate can be normally selected in the terminal from the options available through certain switches. For example, if a terminal has baud rate options as 300, 600 and 900, then anyone of the three baud rates can be selected. The microprocessor should be able to be programmed to the same baud rate. If the baud rates do not match, the whole message will be garbled.

The baud rate (bits per second) adjustment will require the adjustment of time between the transmit/receive of two consecutive bits. The time between two consecutive bits called bit-time can be calculated from the baud rate and the requisite delay can be incorporated. The required time delay between the transmit/receipt of two consecutive bits may be programmed using the delay routine. This can be done by the microprocessor automatically if the desired baud rate is input. Thus the microprocessor can program itself to any baud rate desired.

The CRT terminals use parity bit to check the received data at the lower level. Some terminals transmit and receive odd parity data while others employ the even parity technique. The number of 1s in the code is made even or odd depending on even or odd parity required, using the 7th bit. It should be possible for the microprocessor to cater to both odd and even parity transmission/reception. The software module for parity should generate codes for the required parity. The character should be checked for the same parity at the time of receipt.

7.11 PRINTER INTERFACE

There are more than 1000 types of printers available today. The printers may be dot matrix printer, daisy wheel printer or line printer. Dot matrix printers use print heads which contain wires (pins) arranged in the matrix (tabular) form. These wires are fired to print characters in a certain matrix format (5 × 7, 7 × 7, 9 × 9, etc.).

Daisy wheel printers are slow speed printers but provide high print quality. These printers have a rotating print element (made of either metal or plastic) with a circular series of flexible spokes. The hammer fires the spoke forward to the ribbon and paper, whenever the character in front of the hammer matches with the character to be printed.

In case of line printers, the printing is done line-by-line instead of character-by-character. The line printers may be either band, drum or chain type of printers. A detailed discussion on printers is outside the scope of this book.

The printer interface may be serial or parallel. The serial interface is RS-232C. The printer receives the data bit-by-bit and stores it into an input buffer. The input buffer can normally accommodate characters for one line. On receipt of the print command from the computer (or from the printer when the buffer becomes full) the printer sends the busy signal and starts printing.

In case of parallel interface, the printer has 7 or 8 data lines and some control lines. The computer puts the data to be printed on the data lines and then issues a strobe signal to load the data. The printer stores this data into a buffer and then sends back the acknowledgement signal. Figure 7.84 shows these signals. The printer also sends back the status signals, e.g. Busy, Fault, etc. which are used by the processor for data transfer.

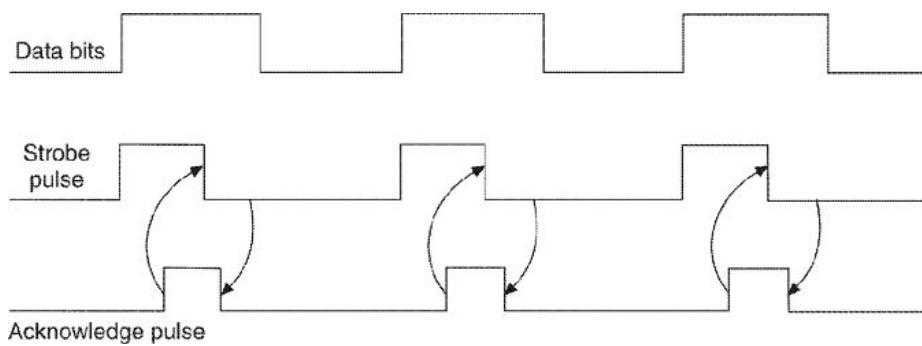


Figure 7.84 Parallel printer interface signals.

The names, functions and the timings of these signals will differ from printer to printer. However most of the printers accept signals and codes as per ASCII (American Standard Code for Information Interchange)

standards. The parallel interface is better than the serial interface since the data transfer is faster as well as simpler and flexible.

In the early 1970s, the printer maker Centronics developed an electrical interface and signal handshaking that allowed parallel transmission to their printers. As this was a fairly simple design, many computer makers of the era adopted it. Many competing printer manufacturers also adopted the Centronics interface and handshake, allowing their printers to work on those computers that provided Centronics-compatible interfaces. By 1981, the Centronics interface was universally accepted. Centronics interface is still quite popular, in spite of a number of new developments. Following is the pin layout of the Centronics printer interface.

<i>Pin no.</i>	<i>Signal</i>	<i>Direction</i>	<i>Description</i>
1		IN	STROBE pulse to read data in. Pulse width must be more than 0.5 ms at the receiving terminal. The signal level is normally high; reading of data is performed at low level of this signal.
2	DATA 1	IN	These signals represent information of the 1st
3	DATA 2	IN	to 8th bits of the parallel data respectively.
4	DATA 3	IN	Each signal is at high level when the data is
5	DATA 4	IN	logical 1 and low when it is logical 0.
6	DATA 5	IN	
7	DATA 6	IN	
8	DATA 7	IN	
9	DATA 8	IN	
10		OUT	Approximately 0.5 ms pulse; low indicates that data is received and the printer is ready for next data.
11	BUSY	OUT	A high signal indicates that the printer cannot receive data. The signal becomes high in the following cases: (1) during data entry, (2) during printing operation, (3) in the “offline” status, (4) during the printer error status.
12	PE	OUT	A high level signal indicates that the printer is out of paper.
13	SLCT	OUT	Indicates that the printer is in the state selected.
14		IN	With this signal being at low level, the paper is fed automatically online after printing.
15	NC		Not used.
16	0V		Logic GND level.
17	CHASISGND	—	Printer chassis GND. In the printer, chassis GND and the logic GND are isolated from each other.
18	NC	—	Not used.
19—	GND	—	“Twisted pair return signal”, GND level.

30			
31	RS	IN	When this signal becomes low, the printer controller is reset to its initial state and the printer buffer is cleared.
32	PAPER	OUT	The level of this signal becomes low when the printer is in “paper end”, “off line” and “error” state.
33	GND	—	Same as with pin numbers 19 to 30.
34	NC	—	Not used.
35			Pulled up to +5 V dc through a 4.7 kW resistance.
36	DATA IN	IN	Data entry to the printer is possible only when the level of this signal is low.

Following signals govern the operation of the printer:

Data Lines

DATA 1 TO DATA 8

Strobe Signal

STROBE

Acknowledge

ACK

The timing diagram illustrates the sequence of events for a data transfer. The **BUSY** signal is high throughout. The **ACK** signal goes low for a minimum duration of $0.5 \mu s$. During this low period, the **DATA** signal is sampled. After the **DATA** signal is sampled, the **STROBE** signal goes high. Vertical lines indicate specific sampling instants.

Figure 7.85 Centronics printer interface timing diagram.

The timing diagram is shown in Figure 7.85. The following signals from the printer may be monitored as printing will be performed only in the absence of these signals.

Error Status

ERROR

Printer out of paper

PF

In addition, Printer Controller and Print Buffer need to be reset in the beginning. This is performed by sending the `INIT` signal to the printer.

In conclusion, the following signals are required for interfacing the Centronics printer to the microprocessor or the computer:

- | | | |
|---|---|--------|
| 1. Data Lines (DATA 1 to DATA 8) | 8 | Output |
| 2. Strobe Signal ($\overline{\text{STROBE}}$) | 1 | Output |
| 3. Acknowledge ($\overline{\text{ACK}}$) | 1 | Input |
| 4. Busy (BUSY) | 1 | Input |

- 5. Error ($\overline{\text{ERROR}}$)
- 6. Paper Exhaust (PE)
- 7. Initialize ($\overline{\text{INIT}}$)

1	Input
1	Input
1	Output

Thus, there will be the requirement of one output port and two output lines and four input lines to interface a Centronics printer to the microprocessor. The hardware interface of the Centronics printer with the microprocessor is shown in Figure 7.86. The flowchart in Figure 7.87 explains the complete working of the microprocessor–printer interface.

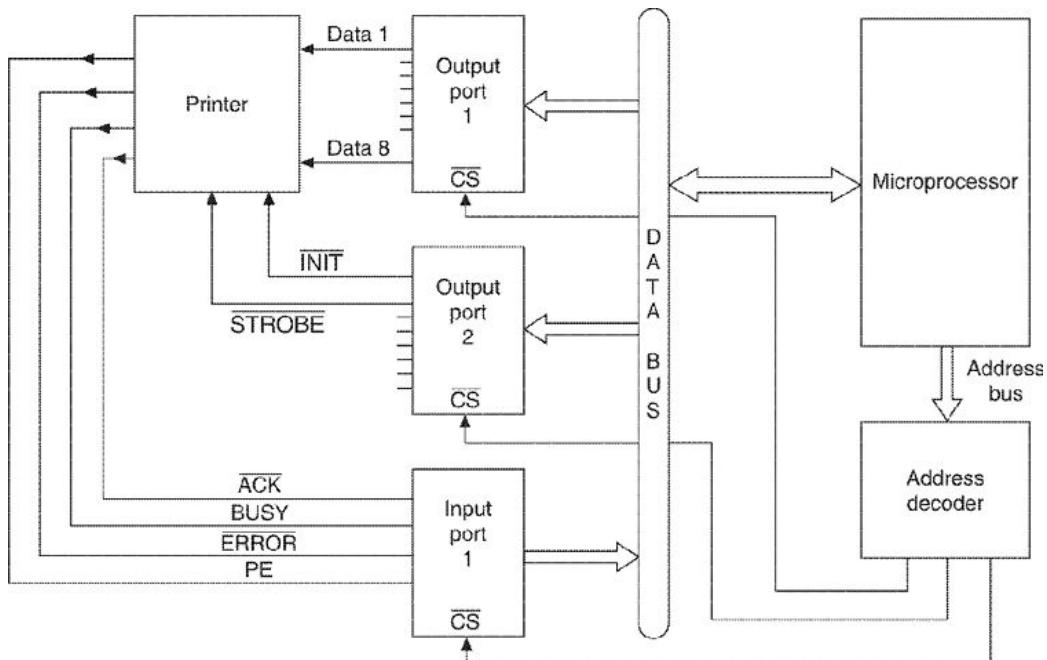


Figure 7.86 Centronics printer interface to microprocessor.

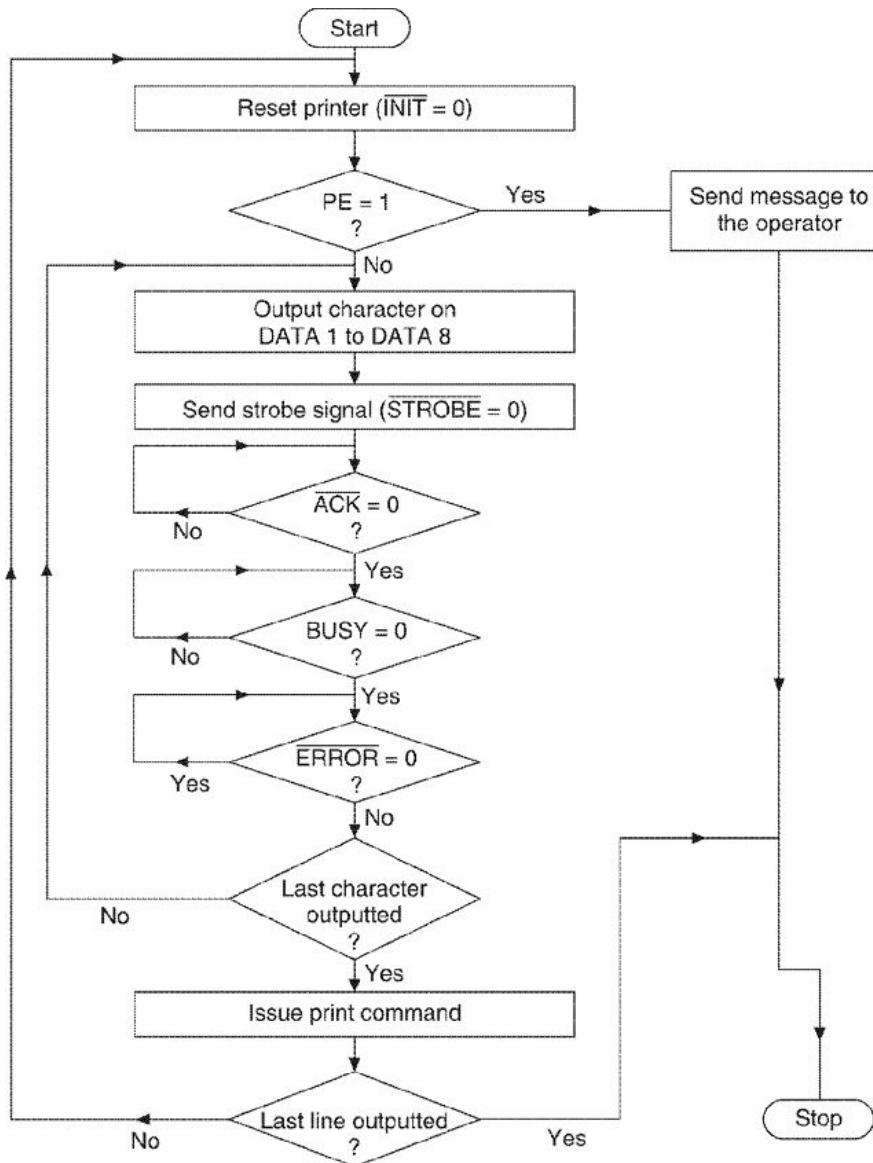


Figure 7.87 Flowchart for Centronics printer interface.

The character codes (in ASCII form) should be stored in some RAM area of the microprocessor. The microprocessor should output the character codes to the printer one-by-one. Line feed should be the last code to be output.

Printers are often capable of executing commands. Such commands are sent by the microprocessor over the data bus. The distinction between ordinary data and commands is accomplished by means of **escape (ESC) codes**. Whenever the microprocessor sends an ESC code, the printer interprets the immediately following code as a command. Such a command may specify the size of the margins, the amount by which the paper advances for each line feed, the desired font, etc.

7.12 CONCLUSION

We have discussed all the facets of microprocessor interfacing in this chapter. Some microprocessors provide on-chip ports; thus PPI may not be required. However, other microprocessors may require port generation through specialized circuits. The interfacing principles and techniques shall remain the same irrespective of any microprocessor configuration.

EXERCISES

1. Write and test the program for the flowchart in Figure 7.73.
 2. Develop interfacing circuits for 10 ports to the 8085/8086 having the following port numbers.

Port nos. — 0F, 1F, 1E, 2C	— Input ports
Port nos. — 3C, EF, 0E, 3D	— Output ports

Interconnect these ports so that the microprocessor can read the data back.
 3. In Example 7.3 regarding system security status and command, the security status and command are not displayed. Using LEDs, extend the design of the system software to include the display of the security status as well as the command of each system.
 4. In an airport arrival hall, the arrival times of flights are displayed in the following format:

Flight No.	Arrival Time
<input type="text"/>	<input type="text"/>

As an example, if the flight IC 7384 is expected at 9:30 pm, it will be displayed as follows:

7384 | 2130

Design a microprocessor-based system that will take the flight number from the keyboard and then display the time from memory. The system will also allow the entering of the changed schedules. A total of four flight details can be displayed at one time. Design a circuit using the 8085/8086 and the 8279 to perform the task.

5. In the ticket counter system at metro railways, explained previously, the system returns to its original status, on encountering any error in software. Extend the system design to convey the error status by a blinking LED.
 6. In the modified ticket counter system at metro railways incorporating the rEADY display, the system will return on encountering an error. Extend the system design to include an

audio alarm which can be reset only by putting off the system.
Study the audio alarm circuit given in standard books.

7. Study the popular analog demultiplexer, digital demultiplexer and DAC chips and redesign the circuit in Figures 7.69 and 7.70, according to the pin configuration.
8. Redesign the circuit in Figures 7.78 and 7.79 with ADC 0800/0809 (which also contains an 8-channel multiplexer) and DAC 0800. Study the working of these chips in TTL Data Book. You may select an analog demultiplexer from TTL Data Bank and incorporate it in the design.
9. In the software program (Example 7.7) for temperature control, the conversion to engineering units has been left out. Study the concept of conversion to engineering units and modify the software to include this. Extend the design to include the display of the present temperature in each floor.
10. The air-flow control action (Example 7.7) requires some finite time, say 2 minutes, before the effect can be felt. During this time, no action needs to be taken. This is the dead time concept. Incorporate the dead time between 2 cycles in the software.
11. Design a fire monitoring and alarm system in a modern 20-floor building. Study the various sensors, and design both hardware and software.

FURTHER READING

Analog Systems for Microprocessors and Minicomputers, Reston Publishing Company Inc., Reston, Virginia, 1978.

Connelly, S.A., *Analog Integrated Circuits*, John Wiley & Sons, New York, 1975.

Krishna Kant, *Microprocessor Based Data Acquisition System Design*, Tata McGraw-Hill, New Delhi, 1987.

Krishna Kant, *Computer Based Industrial Control*, Prentice-Hall of India, New Delhi, 1997.

Leventhal, Lance A., *Intel 8080A/8085 Assembly Language Programming*, A. Osborne McGraw-Hill, Berkeley, California, USA, 1978.

MCS'85 Users Manual, Intel Corporation Ltd., Santa Clara, 1983.

Osborne, Adam and Jerry, Kane, *Some Real Microprocessors*, A. Osborne Associate Inc., Berkeley, California, 1978.

Peripheral Design Handbook, Intel Corporation Ltd., Santa Clara, 1983.

8

SYSTEM DESIGN USING INTEL 8085 AND INTEL 8086 MICROPROCESSORS CASE STUDIES

8.1 INTRODUCTION

In this chapter we shall discuss the various aspects of system design using the 8085 and the 8086 microprocessors. Having done the interfacing of the microprocessor with various subsystems, we find ourselves equipped to take any design problem and solve it. The experience gained in Chapter 7 will help us in designing systems using any microprocessor.

8.2 CASE STUDY—1 A MINING PROBLEM

Figure 8.1 shows the cross-section of a coal mine, where water is getting accumulated. The operator shall start the pump motor when the water reaches the level 'A'. It so happens that the control for the pump motor is at a place from where the water level is not visible. In addition, it is desired to know the rate of the rise of water level and if it is more than the limit fixed, the alarm should be sounded so that the operator can switch on the standby pump and take any other advance action such as evacuating the mine.

The problem looks quite complex *prima facie*, since this is what all mines are faced with. We shall attack this problem in steps from the source, i.e. water level sensing.

8.2.1 Sensors

The application requires a liquid level limit sensor. A probe (working on the conductivity principle) installed at level A will give an indication when the water touches the probe, i.e. when the water level reaches the level A. But how to know the rate of rise in the level?

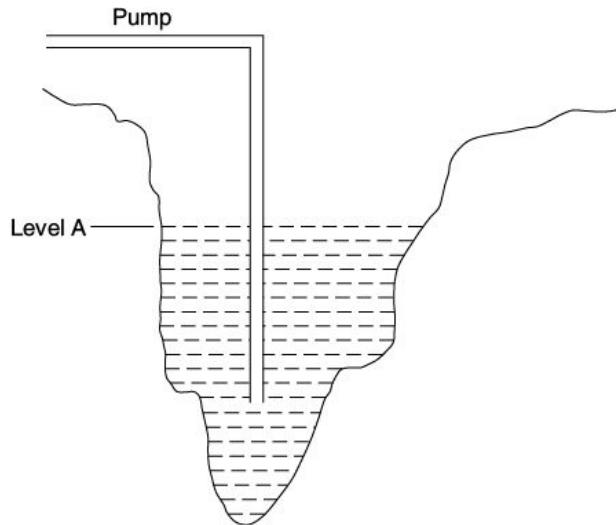


Figure 8.1 Cross-section of a coal mine.

Suppose we have another probe at level B (Figure 8.2) which is less than the level at A. As the water level rises, it will first reach the level B. The probe at level B will give an indication to this effect. When the water level reaches the level A, the probe at that level will give an indication to that effect. If we know the difference between level A and level B and the time taken by water to reach A from B, the rate of rise in the level can be derived.

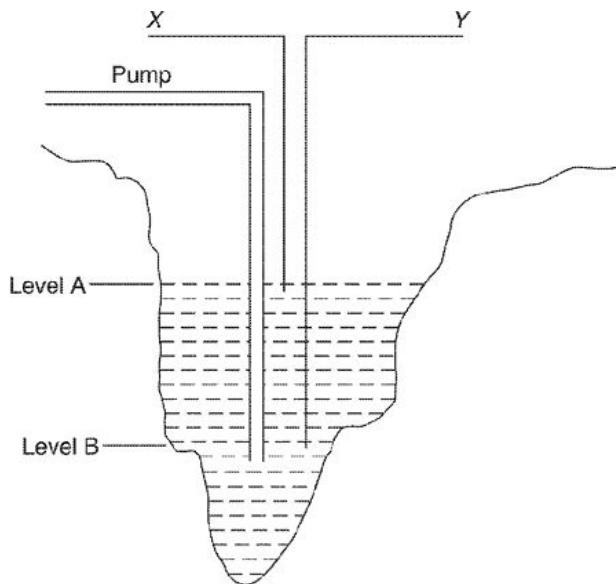


Figure 8.2 Adding another probe at level B, at a depth less than the level at A.

The sensing circuit is shown in Figure 8.3. R_1 and R_2 are the two relays. The circuit and pit grounds are common. When water is below the level B, both the transistors T_1 and T_2 are not conducting. The level A and level B outputs remain at 0 volt. When water reaches the level B,

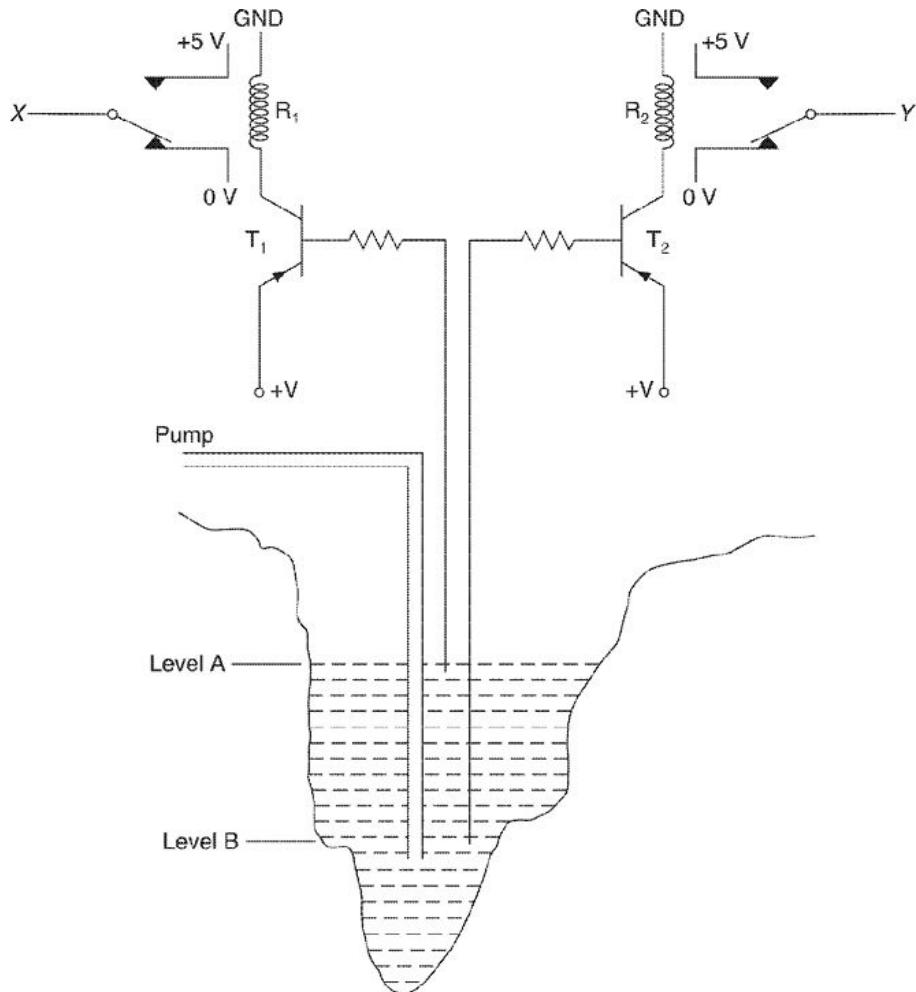


Figure 8.3 Water level sensing circuit.

the transistor T_2 will conduct, which will enable relay R_2 to operate and due to which the level B output will be switched to 5 volts. Similarly, when water reaches the level A, both the transistors will conduct and both level A and level B output points will be at 5 volts.

8.2.2 Multiplexer and Data Converter

Since there are two input points, marked X and Y , which are coming from level A and B sensors respectively, we would require a two-channel multiplexer to read the data alternatively from the two channels. One may use one of the available multiplexer chips. The same purpose can,

however, be achieved by developing a simple two-level switch as shown in Figure 8.4.

The time taken by the water to reach level A from level B will be in seconds. Thus, the sampling time of 10 milliseconds (100 samples per second) will be sufficient for this application. For such a low variation rate, a sample-and-hold circuit will not be necessary, since there is no need to hold the data which is changing at a very slow rate, and which we are able to otherwise sample at a very fast rate comparatively.

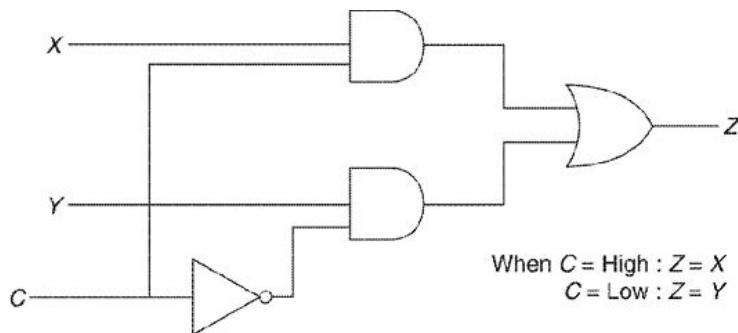


Figure 8.4 A simple two-level switch.

The signals available from the two probes X and Y (Figure 8.3) will indicate the presence or absence of water at level A and B respectively. When the signal is high (+5 V), the water is present at the level and when the signal is low (0 V), water is yet to reach the level. When the water is below level B, both X and Y are at 0 V; when water touches level B, Y comes to 5 V but X remains at 0 V; when water reaches level A, X also goes to 5 V. This is shown in Figure 8.5. Since X and Y are in digital form (High or Low), these can be directly read by the microprocessor for processing.

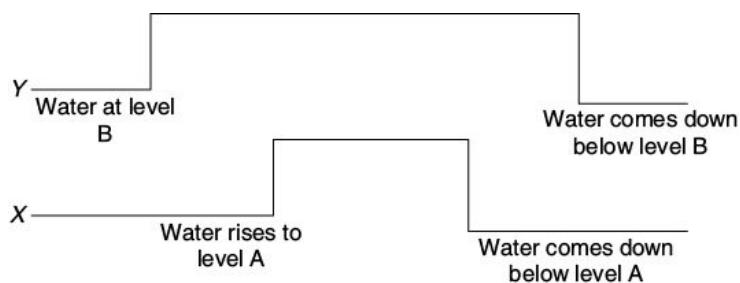


Figure 8.5 Timing diagram for water level sensor.

8.2.3 I/O Ports

The interfacing of X and Y probes to a microprocessor will require two ports—one input port to read the data from X and Y through the multiplexer and one output port to give the address information to the multiplexer in order to select either X or Y . The two ports can be created

by using a hardware circuit. A specialized peripheral interface chip can also be used. In addition, another output port will be required for annunciation. As mentioned before, two types of annunciations are required.

1. When the water reaches level A.
2. When the rate of increase in the level is more than the predefined limit.

One bit of input port (Figure 8.6) can be connected to the multiplexer to read X or Y. Similarly, one bit of the output port 1 can be used to furnish the address to the multiplexer and two bits of output port 2 can be used for setting up Alarm 1 and Alarm 2 as mentioned above.

The I/O ports can be implemented using the 8255 Programmable Peripheral Interface (PPI) chip.

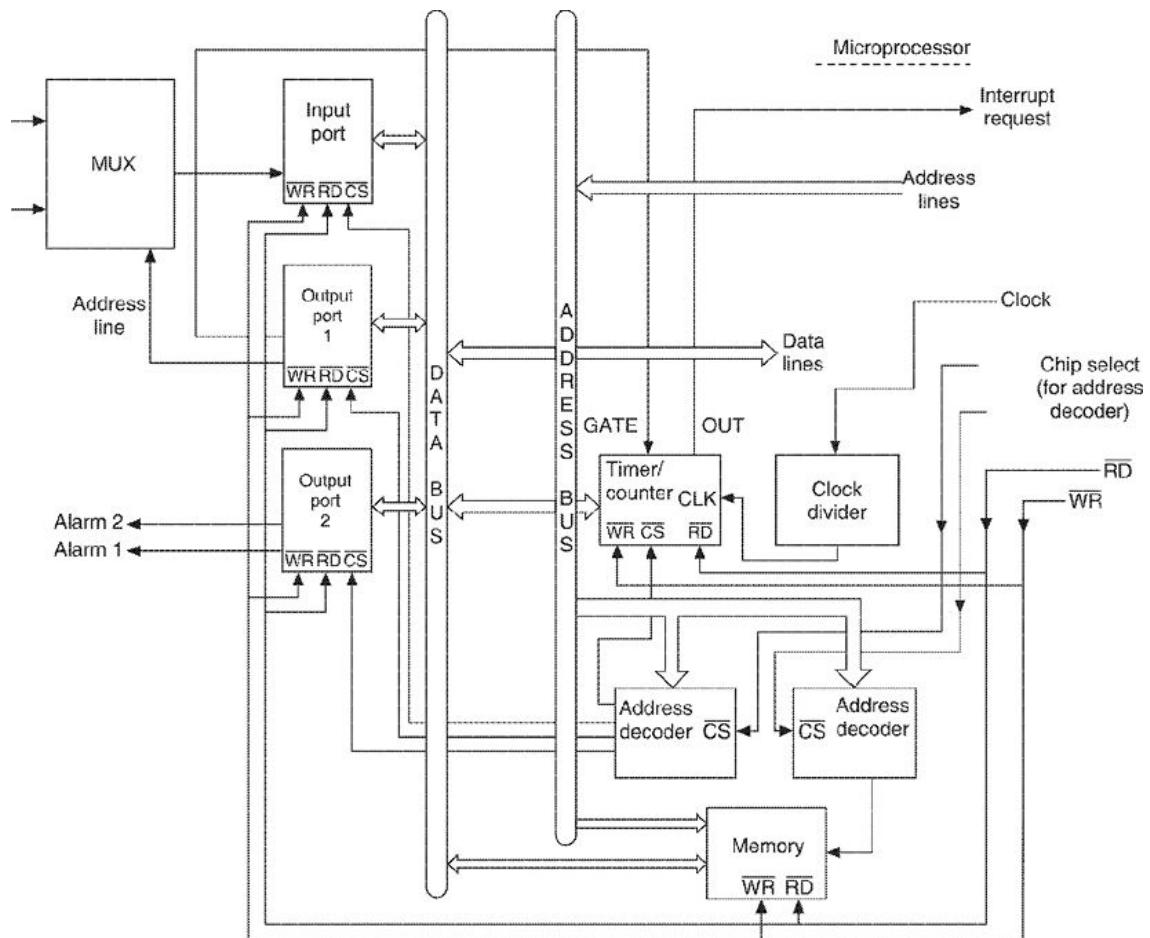


Figure 8.6 Block diagram of the microprocessor-based water level monitoring system.

8.2.4 Timer Function

Alarm 2 should be ON when the rate of rise of the level is more than the predefined limit. The predefined limit is stored in memory. The microprocessor will put Alarm 2 to ON if the rate of the rise of level limit is violated. The rate of rise of the level is calculated by the time taken by water to reach level A from level B. This time cannot be calculated by the software as the microprocessor cannot scan the probes and calculate the time at the same instant. Thus an external timer/counter chip will be needed. The microprocessor will load the maximum count in the counter initially and will activate the timer/counter when the water level reaches B. The microprocessor will then continue scanning the probe for level A and the timer/counter will be decrementing the count simultaneously. When the water touches level A, the microprocessor stops the timer/counter and reads the count and calculates the time elapsed.

The 8253 chip may be used for timer/counter function. One of the lines of output port may be used to initiate the timer/counter when water reaches level A. Since 8253 works on 2 MHz clock frequency, the microprocessor frequency may be divided using a clock divider circuit.

We may have to use two timer/counters of the 8253 chip. The first one will generate 1 second clock pulse, which will be input to the second counter. The output of the second counter may be connected to one of the interrupt pins of the microprocessor. The interrupt will be caused when the counter reaches 0. The interrupt servicing routine may implement a software counter to supplement the two timer/counters of the 8253 chip.

8.2.5 Alarm Circuit

The alarm may be both audio and visual. For audio alarm, a circuit using 7474 and 555 IC along with the speaker may be used as shown in Figure 8.7. The same line can be used to drive an incandescent lamp for visual indications. When the input line is high, the alarm is ON; and the alarm goes OFF, when the input line is low. There is an acknowledge switch associated with the alarm circuit. When the alarm is ON and the operator acknowledges it by pressing the acknowledge switch, the alarm goes OFF.

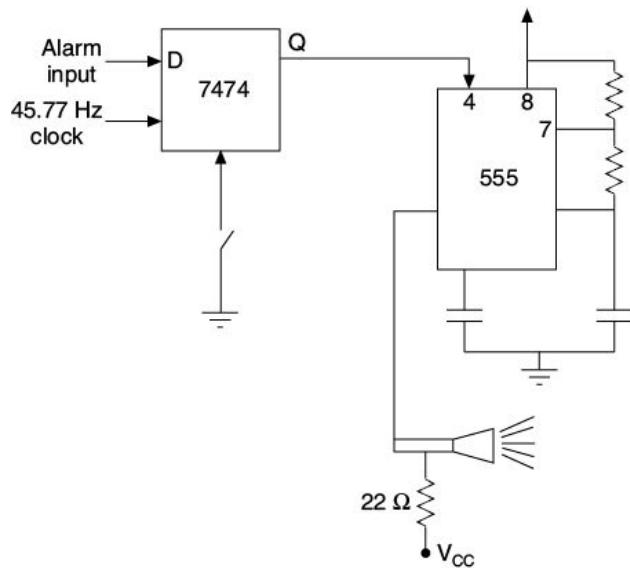


Figure 8.7 Audio alarm circuit.

8.2.6 Software

The software resident in the memory of the microprocessor will do the following tasks in sequence.

- (a) Initialization
- (b) Reading Y
- (c) Initiating timer/counter
- (d) Reading X
- (e) Reading the contents of the timer/counter
- (f) Calculating the rate of rise of level
- (g) Setting Alarm 1 or/and Alarm 2.

Figure 8.8 shows the flowchart. Initialization of the programmable ports and the timer/counter involves the determination of the control word and transferring them to these programmable chips so that the desired mode and ports are set.

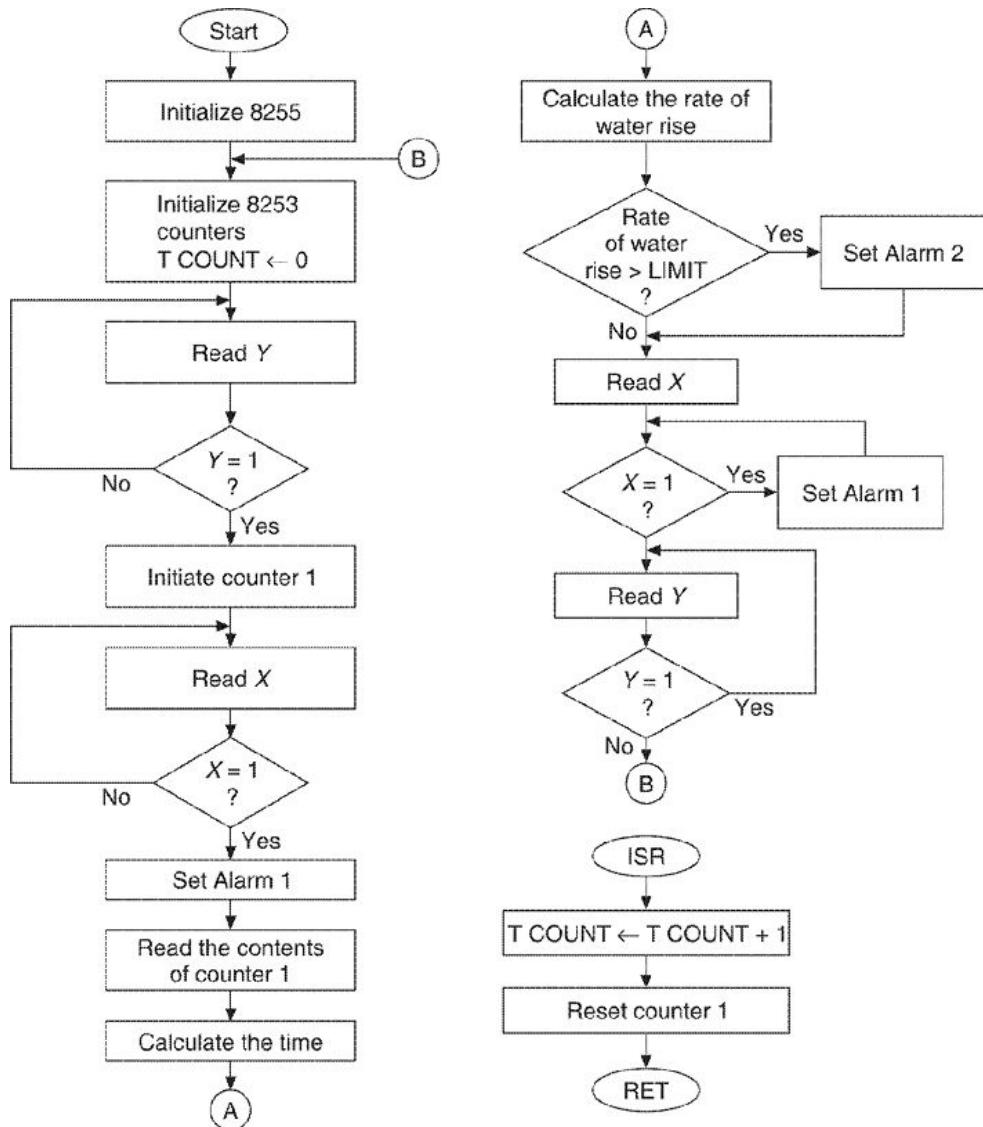


Figure 8.8 Flowchart for water level monitoring system.

Let us now solve this problem first by using the 8085 and then by using the 8086.

8.2.7 Using the 8085

Figure 8.9 shows the 8085, the 2716/6116 memory chips, the 8255 PPI, the timer/counter 8253 and the analog multiplexer AD 7502, all integrated to form the system for this application. The input clock frequency is 6 MHz. Thus, the internal clock frequency and the frequency at CLK OUT will be 3 MHz.

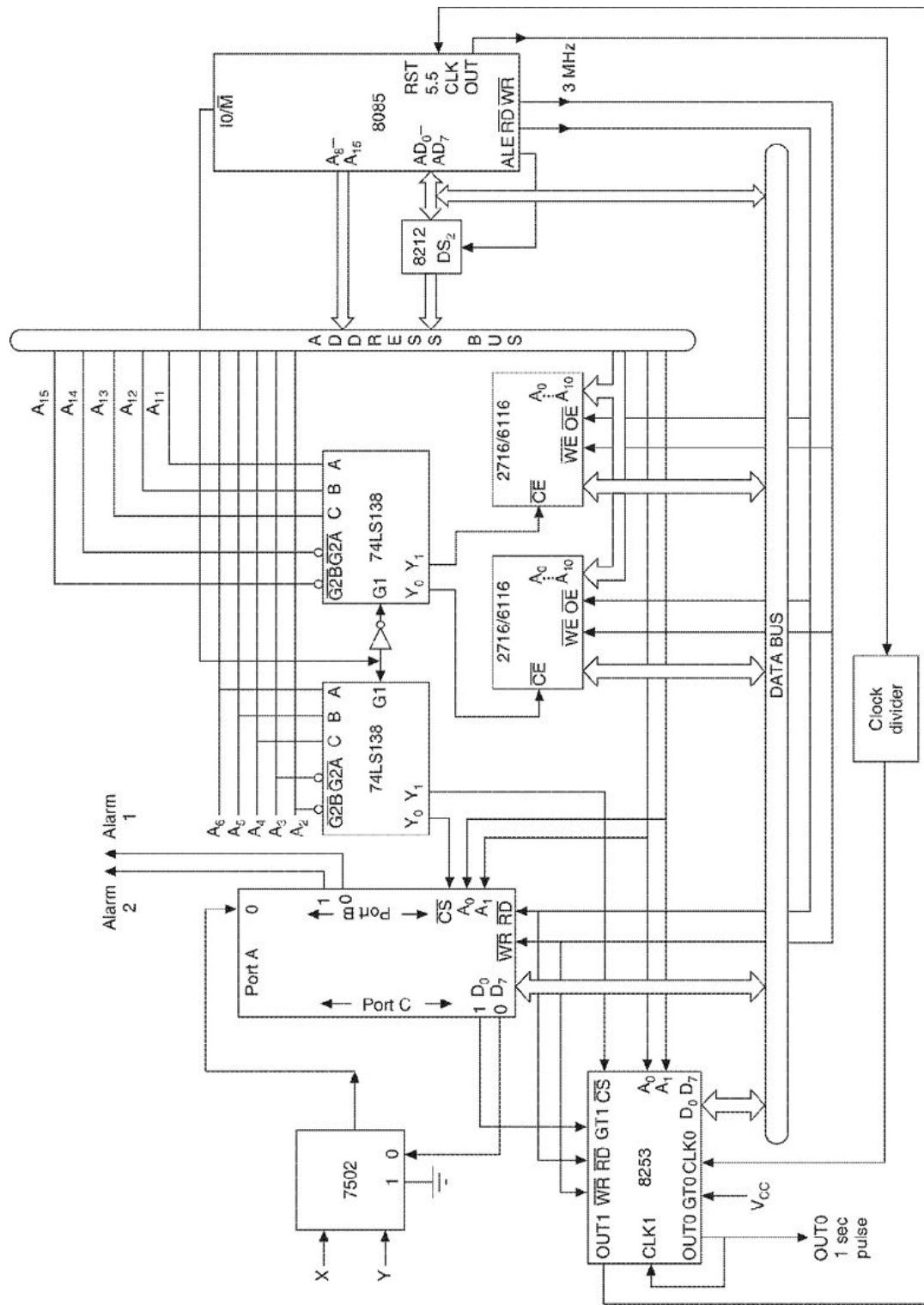


Figure 8.9 Water level monitoring system using the 8085.

It should be noted that the timer/counter 8253 does not accept the input clock of frequency more than 2 MHz. That is why the CLK OUT of the 8085 is divided by 256 through two 7493s to get 11.7 kHz (Figure 8.10) clock. The time-period of this clock is 0.085 millisecond. This is fed to Counter 0 of the 8253 to derive 1 second clock pulse. The 1

second pulse is used to determine the time through Counter 1. The total time that can be measured is thus 65535 seconds, i.e. 18 hours, 12 minutes and 15 seconds. The OUT1 signal from Counter 1 is connected to RST 5.5 interrupt to reset the counter. The ISR RST 5.5 increments a software counter TCOUNT every time an interrupt occurs. One count of TCOUNT is equivalent to 18 hours 12 minutes and 15 seconds. Thus, if $TCOUNT > 0$ then water is rising very slowly and Alarm 2 need not be initiated. The interfacing of 2716/6116 memory chips, the timer/counter 8253 and the 8255 with the 8085 has been dealt with in detail in the previous chapters.

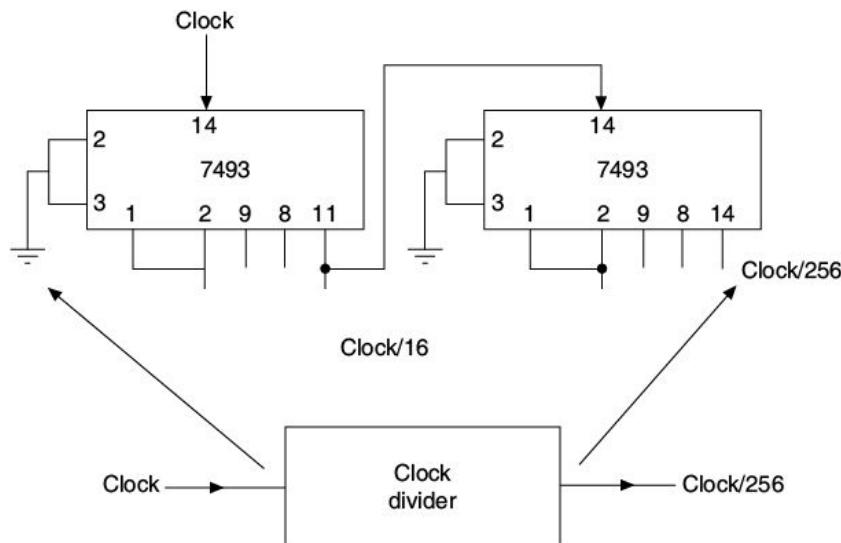


Figure 8.10 Clock divider circuit.

The memory address space is from 0000H to 0FFFH, i.e. 4 KB. The 8255 port numbers are from 00H to 03H and the 8253 port numbers are from 04H to 07H as derived from the \overline{CS} signals for these chips in the diagram.

Port A of Intel 8255 is configured as input port. Bit 0 of port A is connected to the output of the multiplexer. Port B and Port C are configured as output ports. Bit 0 and Bit 1 of Port B are connected to Alarm 1 and Alarm 2 respectively. Bit 0 of Port C is used as address line to multiplexer whereas Bit 1 is used for providing Gate Signal to Counter 1 for its initiation.

The 8255 initialization

Port Numbers—

Port A	— 00
Port B	— 01
Port C	— 02

Control Register – 03

Port A = Input

Port B = Output

Port C = Output

Mode = 0

Control Word = 1 0 0 1 0 0 0 0 = 90H

The 8253 initialization

Port Numbers—

Counter 0 – 04

Counter 1 – 05

Counter 2 – 06

Control Register – 07

Counter 0 –

The counter is utilized to generate a 1 second pulse. The mode is therefore 2, i.e. rate generator. It is selected as the binary counter and both LSB and MSB of count value are loaded into the counter. For 0.085 millisecond clock, the count value will be 11765. It will be represented as 2DF5H.

Control Word = 0 0 1 1 0 1 0 0 = 34H

Counter 1 –

The counter is utilized to count the 1 second pulses during the time the water reaches the level A from level B. The mode is 0, i.e. interrupt on the terminal count. It is selected as the binary counter and both the low-order and high-order bytes of count value are loaded into the counter. The count value is the maximum value in 16 bits, i.e. FFFFH.

Control Word = 0 1 1 1 0 0 0 0 = 70H

The initiation of the timer/counter involves making the gate input high. Whereas the gate input for Counter 0 is made permanently high, for Counter 1 it is done when the water reaches the level B.

The rate of rise may be directly calibrated to time, i.e. the number of clock periods. In fact, the limit for the rate of rise is stored in the microprocessor's memory in terms of the number of clock periods. The alarms are set by making the corresponding bits as 1.

;

; Intel 8085 software for water level monitoring in a coal mine.

; Main program

START: MVI A, 90H; Initialize Intel 8255

OUT	03H	
MVI	A, 34H; Initialize Counter 0	
OUT	07H	
MVI	A, 70H; Initialize Counter 1	
OUT	07H	
MVI	A, F5H; Load Counter 0 with lower byte of	
count		
OUT	04H	
MVI	A, 2DH; Load Counter 0 with higher byte of	
count		
OUT	04H	
LOOP0:	MVI	A, FFH; Load Counter 1 with lower
byte of count		
OUT	05H	
OUT	05H; Load Counter 1 with higher byte of count	
MVI	C, 00H; Register C used for TCOUNT	
; Check if water has reached level B.		
MVI	A, 00H	
OUT	02H	
LOOP1:	IN	00H
CPI	01H	
JNZ	L00P1	
MVI	A, 02H; Initiate Counter 1	
OUT	02H	
; Check if water has reached level A.		
MVI	A, 01H; Initiate MUX channel = 1	
OUT	02H	
L00P2:	IN	00H
CPI	01H	
JNZ	L00P2	
MVI	A, 01H; Set Alarm 1	
OUT	01H	
MVI	A, 00H; Inhibit Counter 1	
OUT	02H	
MOV	A, C; Check if TCOUNT > 0	
ANA	A; If yes then limit not violated.	
JNZ	L00P3	
IN	05H; Read Counter 1 (Lower byte)	
MOV	E, A	
IN	05H; Read Counter 1 (Higher byte)	

```

MOV      D, A
MVI      A, FFH; Subtract the contents of DE from
          ; FFFFH to get the actual count
SUB     E
MOV      E, A
MVI      A, FFH
SBB     D
MOV      D, A; Actual count is in D-E register pair
LDA      LIMIT1; Load the lower byte of limit
SUB     E
LDA      LIMIT2; Load the higher byte of limit
SBB     D

JP      LOOP3
MVI      A, 02H; Limit violated, so set Alarm 2
OUT     01H

L00P3:   MVI      A, 01H; Initiate MUX channel 1
          OUT     02H
          IN      00H
          CPI     01H
          JNZ    L00P4
          MVI      A, 01H; Set Alarm 1
          OUT     01H
          JMP    L00P3
; Check whether water has come below level B.

L00P4:   MVI      A, 00H
          OUT     02H
          IN      00H
          CPI     01H
          JZ     L00P4
          JMP    L00P0
;

; RST 5.5 Interrupt Servicing routine for resetting Counter 1.
;
RST5.5:  INR      C
          MVI      A, FFH
          OUT     05H
          OUT     05H
          MVI      A, 02H
          OUT     02H
          RET

```

```
LIMT1 DB      —; Lower byte of limit  
LIMT2 DB      —; Higher byte of limit  
END
```

8.2.8 Using the 8086

Figure 8.11 shows the 8086-based system for the application consisting of the 8255 PPI, the 8253 timer/counter, the analog multiplexer AD7502 interfaced to 2716/6116 memory chips and the 8086 microprocessor.

The clock frequency of the 8086 microprocessor is assumed to be 5 MHz. Thus, the frequency of the PCLK output will be 2.5 MHz. This is divided by 256 using two 7493s (Figure 8.10) to yield the clock frequency of 9.765 kHz. The time period of this clock is 0.1024 millisecond. It is fed to counter 0 of the 8253 to derive 1 second clock pulse. The mode is therefore 2, i.e. the rate generator. It is selected as binary counter and both low- and high-order bytes of count value are loaded to the counter. For 0.1024 ms clock, the count value will be 9765. It is represented as 2625H.

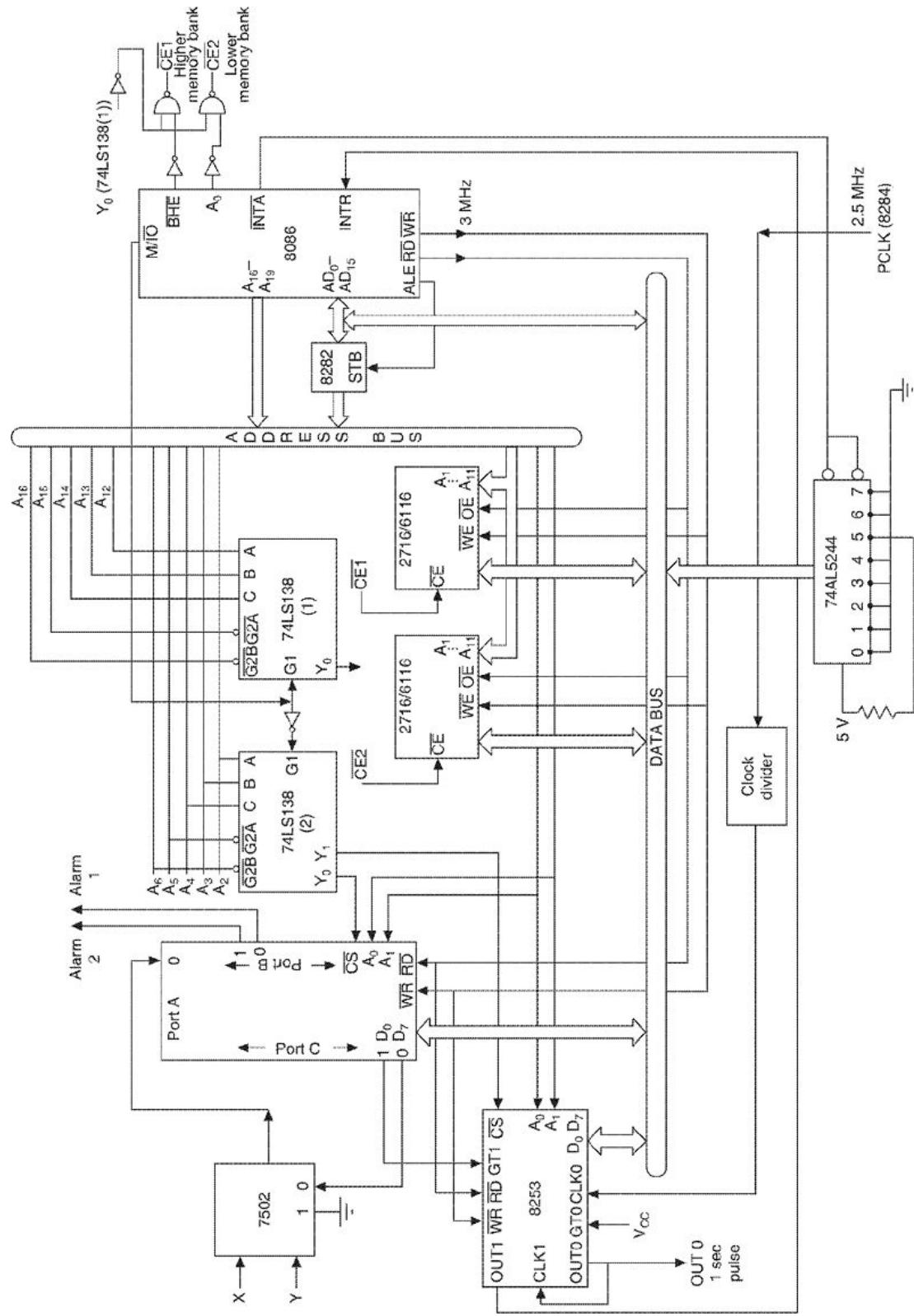


Figure 8.11 Water level monitoring system using the 8086.

The 1 second clock pulse is used to determine, through counter 1, the time taken by the water to reach level A from level B. The mode is 0, i.e. interrupt on the terminal count. It is selected as binary counter and both low- and high-order bytes of count value are loaded to the counter 1. The

count value is the maximum value in 16 bits, i.e. FFFFH. The total time that can be measured is, thus, 65535 seconds, i.e. 18 hours, 12 minutes and 15 seconds. The OUT1 signal from Counter 1 is connected to INTR (Type 20H) interrupt to reset the counter.

The initiation of timer/counter involves making the gate input permanently high, for Counter 0. For Counter 1, it is done when the water reaches the level B. The rate of rise may be directly calibrated to time, i.e. the number of clock periods. In fact, the limit for the rate of rise is stored in memory in terms of the number of clock periods.

The alarms are set by making the corresponding bits 1. Memory chips have been connected for the address space 0000H to 0FFFH, i.e. 4 KB. The memory is divided into high and low memory bank of 2 KB each. The chip enable signals ($\overline{CE1}$ and $\overline{CE2}$) for these chips are generated by combining the output Y_0 of 74LS138 (1) with A_0 and BHE in the same way as shown in Figure 5.39 (Chapter 5). The 8255 port numbers are from 00H to 03H and the 8253 port numbers are from 04H to 07H as derived from \overline{CS} for these chips in the diagram.

The 8255 port configurations are:

A Input Bit 0—connected to the output of the multiplexer

B Output Bit 0—connected to Alarm 1

 Bit 1—connected to Alarm 2

C Output Bit 0—connected to the address line of AD
7502

 Bit 1—connected to GATE 1 of the 8253

;

; Intel 8086 software for water level monitoring in a coal mine.

;

; Main Programs

;

DATA SEGMENT

; TCOUNT = No. of times Counter 1 has completed count of FFFFH without water

; reaching Level A from Level B. If TCOUNT > 0 then water is rising very slowly and

; the limit is not violated.

;

; LIMIT = Rate of rise of water level. Defined in terms of time taken for water to reach

```

; Level A from Level B, i.e. No. of clock periods:
;

        LIMIT      DW      -
        TCOUNT    DB      0
        DATA ENDS
S-STACK SEGMENT
        DW 100 DUP (0)
        T-STACK LABEL WORD

        S-STACK ENDS

;
; Declare ISR CNTIRS in other module.
;
PROCEDURES SEGMENT PUBLIC
        EXTRN CNTIRS: FAR
PROCEDURES ENDS
        PUBLIC TCOUNT
CODE SEGMENT

ASSUME CS: CODE, DS:DATA, SS: S-STACK
;; Initialize segment registers.
;

START:    MOV      AX, DATA
        MOV      DS, AX
        MOV      AX, S-STACK
        MOV      SS, AX
        MOV      SP, OFFSET T-STACK
;

; Initialize Intel 8255—Port A = Input, B = Output, C = Output, Mode
= 0
; Control Word = 1 0 0 1 0 0 0 0 = 90H
; Intel 8255 Port Nos.—A = 00H, B = 01H, C = 02H, Control word =
03H
;
        MOV      AL, 90H
        OUT      03H, AL
;

; Insert the address of ISR CNTIRS in interrupt vector table. Since the
vector type is 20H,

```

```

; the offset and segment addresses are to be stored at 20Hx4 = 0080H
as offset at 0080H,
; 0081H and segment address at 0082H, 0083H.
;

MOV      AX, 0000H
MOV      ES, AX
MOV      WORD PTR ES: 0080H, OFFSET
CNTIRS
MOV      WORD PTR ES: 0082H, SEG CNTIRS
;

; Initialize Counter 0, mode = 02, i.e. Rate generator, Binary counter.
; Control Word = 0 0 1 1 0 1 0 0 = 34H
;

MOV      AL, 34H
OUT     07H, AL
MOV      AL, 25H; Lower byte of count
OUT     04H, AL; for Counter 0
MOV      AL, 26H; Higher byte of
OUT     04H, AL; count for Counter 0
;

; Initialize Counter 1, mode = 0 i.e. interrupt on the terminal count,
Binary counter
; Control Word = 0 1 1 1 0 0 0 0 = 70H
;

MOV      AL, 70H
OUT     07H, AL
;

; Load count FFFFH in counter 1.
;

LOOP0:   MOV      AL, FFH
        OUT     05H, AL
        OUT     05H, AL
        MOV      TCOUNT, 00H
;

; Check water level at B.
;

MOV      AL, 00H; Initiate MUX channel 0
OUT     02H, AL
LOOP1:   IN      AL, 00H; (AL) □ (Y)
        CMP     AL, 01H; Check if (Y) = 1?

```

```

        JNZ           LOOP1
;
; Water is at Level B. Initiate Counter 1 to determine the rate of rise of
water.
;
; MOV          AL, 02H
; OUT         02H, AL
;
; Check for water at Level A.
;
; MOV          AL, 01H; Initiate MUX channel
1
        OUT          02H, AL
LOOP2:   IN           AL, 00H; (AL) □ (X)
        CMP          AL, 01H; Check if (X) = 1?
        JNZ          LOOP2
;
; Water has reached Level A. Set Alarm 1.
;
; MOV          AL, 01H
; OUT         01H, AL
; Inhibit Counter 1 by making GT1 = 0.
;
        MOV          AL, 00H
        OUT          02H, AL
;
; Calculate the rate of rise of water to determine if the limit is
violated. If TCOUNT > 0
; then the limit is not violated.
        MOV          BL, TCOUNT
        CMP          BL, 00H
        JA           LOOP3;
; TCOUNT = 0
        IN            AL, 05H; Read Counter 1 (lower byte)
        MOV          CL, AL
        IN            AL, 05H; Read Counter 1 (higher byte)
        MOV          CH, AL
;
; Now CX contains (FFFFH-Count). Subtract CX from FFFFH to get
count.
;
        MOV          AX, FFFFH

```

```

        SUB      AX, CX; (AX) = count
        CMP      AX, LIMIT
        JL       LOOP3; (AX) < LIMIT
;
;
; Limit violated. Set Alarm 2.
;
        MOV      AL, 02H
        OUT     01H, AL
;
; Again check for water level at Level A.
;
        LOOP3:   MOV      AL, 01H; Initiate MUX Channel
1
        OUT     02H, AL
        IN      AL, 00H; (AL) □ (X)
        CMP      AL, 01H
        JNZ     LOOP4
;
; Water still at Level A. Again set Alarm 1.
;
        MOV      AL, 01H
        OUT     01H, AL
        JMP     LOOP3
;
; Water below Level A. Check for water at Level B.
;
        LOOP4:   MOV      AL, 00H; Initiate MUX Channel
0
        OUT     02H, AL
        IN      AL, 00H; (AL) □ (Y)
        CMP      AL, 01H
        JZ      LOOP4
        JMP     LOOP0
        NOP
CODE ENDS
DATA SEGMENT PUBLIC
EXTRN TCOUNT
DATA ENDS

```

```

PUBLIC CNTIRS
PROCEDURES SEGMENT PUBLIC
    CNTIRS PROC FAR
        ASSUME CS: PROCEDURES; DS: DATA
; Increment TCOUNT
;
    INC      TCOUNT
;
; Reset counter 1 and load FFFFH.
;
    MOV      AL, FFH
    OUT     05H, AL; lower byte
    OUT     05H, AL; higher byte
    MOV      AL, 02H
    OUT     02H, AL
    IRET
CNTIRS ENDP
PROCEDURES ENDS

```

EXERCISES

1. In the system design, the multiplexer is really not required and inputs X and Y can be directly read through the input port. Modify the circuit design and the software.
2. The system design has the facility for alarm annunciation. The operator has to start the pump on receiving the audio/visual alarm. Modify the design for automatically starting and stopping the pump. Write the software for the same.
3. When the water has reached Level B, the program checks for water at Level A continuously with the assumption that the water level is rising. Suppose the water level reaches B and then goes below Level B. In such a case, the program will be in loop at LOOP2 label. Modify the program to tackle such a situation.

8.3 CASE STUDY 2—TURBINE MONITOR

In a power plant, it is the turbine which drives the generator to generate electricity. In thermal power plants, the turbine is rotated by the high pressure steam. In fact there are three turbines in the order of decreasing pressure, namely the High Pressure (HP) turbine, the Intermediate

Pressure (IP) turbine, and the Low Pressure (LP) turbine. The shaft of the LP turbine is connected to the generator. The steam from the boiler is fed to the HP turbine. The exhaust from the HP turbine is reheated and fed to the IP turbine and the exhaust from the IP turbine is fed to the LP turbine (without reheating). Figure 8.12 shows the block diagram of the turbine system in a thermal power plant. Following are the important parameters which should be monitored at regular intervals for safe and efficient working of the turbine.

1. Main steam pressure
2. Reheat bowl steam pressure
3. Condenser vacuum
4. Generator output voltage
5. Generator output current
6. Shaft speed

The steam flow in the turbine is controlled by controlling the position of the Main Stop Valve (MSV), the Steam Control Valve (CV), the Reheat Stop Valve (RSV) and the Intercept Valve (ICV). As shown in the figure, the MSV and CV control the flow of steam in the HP turbine, and the RSV and ICV control the flow of steam in the IP turbine. The actuating signals are sent to the actuators of these valves by the controller not shown in the figure. The actuators change the position of the valves depending on the actuating signal. The position of a valve is available at the output of the actuator. Thus, in addition to the above six parameters, the following four parameters corresponding to the valve positions should also be monitored.

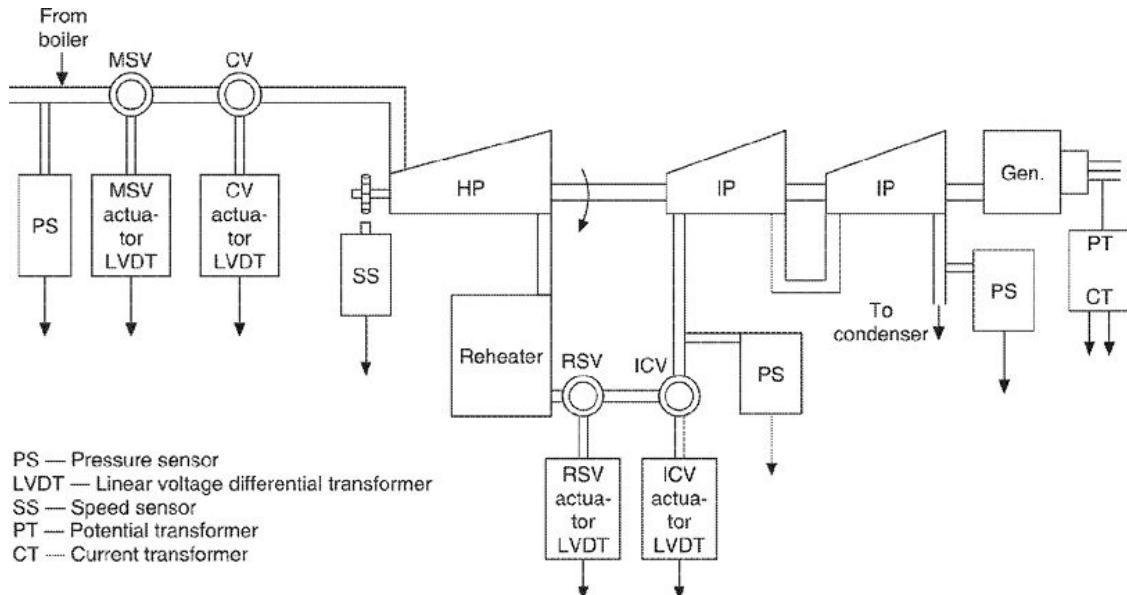


Figure 8.12 Block diagram of a turbine system in a typical thermal power plant.

1. Main Stop Valve (MSV) position
2. Steam Control Valve (CV) position
3. Reheat Stop Valve (RSV) position
4. Intercept Valve (ICV) position

For a 210 MW steam turbine, the typical values for the five parameters are

1. Main steam pressure	= 150 kg/cm ²
2. Reheat bowl steam pressure	= 34.55 kg/cm ²
3. Condenser vacuum	= 78 mm Hg
4. Generator output	= 15.75 kV, 9050 A
5. Shaft speed	= 3000 rpm

Our task is to design a microprocessor-based system for turbine monitoring, i.e. logging and printing of these parameters at regular intervals. It is desired that the current values of the parameters may be printed only when there is a change in any of them and also the current time should be printed along with these values.

8.3.1 Measurement Transducers

We need pressure transducers to measure the steam pressures and condenser vacuum. A speed sensor will be needed for the shaft speed,

and the current/voltage transformers will be needed for the generator output. The actuators will produce the corresponding signals for valve positions.

Steam pressure transducers

Conventional sensors like Bourdon tubes, bellows, etc. are used for steam pressure measurement. However, newer sensors like strain gauges can also be used to measure the steam pressure. When a wire is stretched within its elastic limit, there will be an increase in its length and decrease in its diameter. Thus, the resistance of the wire changes due to the strain produced. This is called the *piezoresistive effect*, and is the basic principle behind the strain gauge transducers.

Strain gauges are popular transducers for pressure measurement. The strain measuring circuit consists of a bridge as shown in Figure 8.13. One arm of the bridge contains the strain gauge while the other arms have standard resistors of equal resistances, as that of the gauge resistance in the unstrained condition. The current through the bridge arm measures the strain.

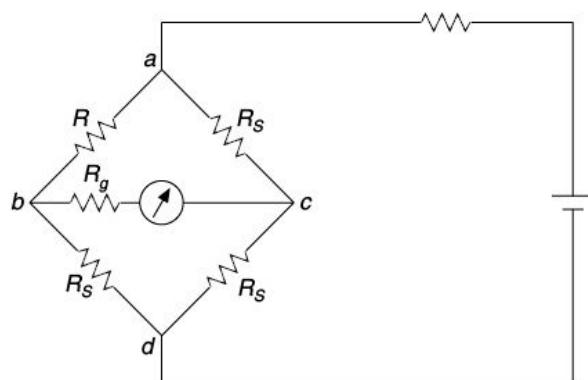


Figure 8.13 Strain measuring circuit.

The output from the strain gauge is conditioned, by a signal conditioning circuit, to make it suitable for ADC input.

Condenser vacuum transducer

A pressure transducer whose range varies from negative to positive pressure, will be required to measure the condenser vacuum. Some conventional sensors like bellows, Bourdon tubes, etc. have this range. The semiconductor sensors can also be used.

Control valve position sensor

To measure the valve position in case of MSV, ICV, RSV and CV valves, LVDTs (linear voltage differential transformer) may be connected to the

valves as part of the actuators. The outputs from these LVDTs are proportional to the position of the control valves.

Linear variable differential transformer (LVDT)

It is used to measure the linear displacement. It is basically a transformer with one primary and two secondary coils with a movable core between them. The schematic diagram of the LVDT is shown in Figure 8.14.

The secondary coils are identical and positioned symmetrically with respect to the primary coil. The secondary coils are connected in phase opposition. The movable core is connected

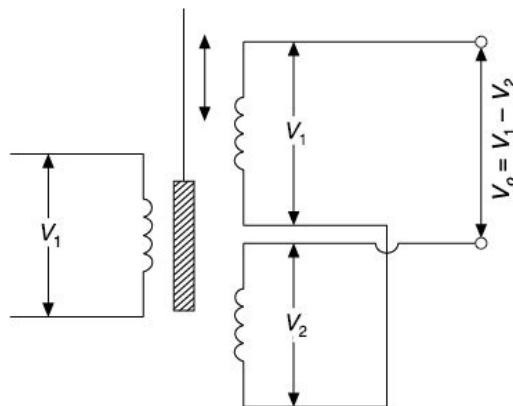


Figure 8.14 Linear variable differential transformer as position sensor.

to the member whose displacement is to be measured. LVDT transducers have a range from 0.001 to several inches. However, the detection of a 10 micro inch displacement can also be made by certain special types of LVDTs.

Magnetic shaft speed sensor

The shaft speed sensor measures the number of rotations made by the shaft. As shown in Figure 8.15, the other end of the shaft is connected to a tooth wheel. The teeth are made of ferromagnetic material. The speed detector contains a permanent magnet with coils wound around. When a ferromagnetic tooth (e.g. magnetic steel) on the rotating shaft passes in the proximity of a permanent magnet, the lines of flux of this magnet cut across the coil winding,

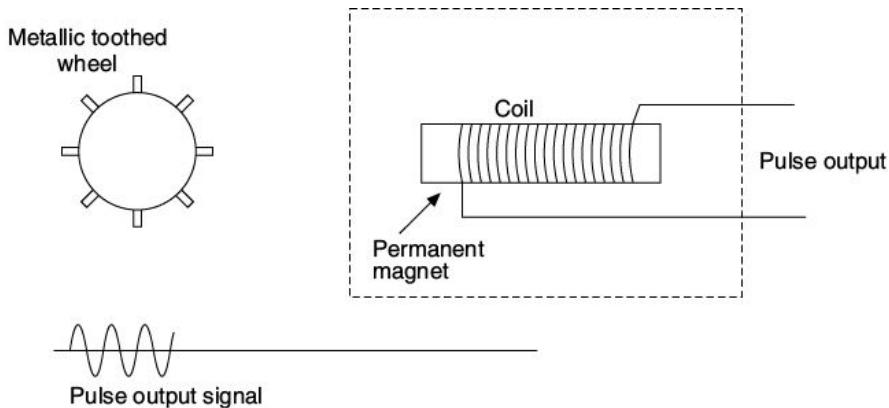


Figure 8.15 Magnetic shaft speed sensing.

thereby inducing an electromagnetic force in the coil. An output pulse is created when a tooth passes in the proximity of the permanent magnet housed in the speed detector. The number of output pulses are, therefore, equal to the number of teeth in the wheel. These pulses should be counted to determine the shaft speed.

Photoelectric shaft speed transducers

These transducers may also be used for measuring the shaft speed. The scheme is shown in Figure 8.16. It consists of a shutter disc mounted on the shaft which rotates between a light source and a light detector. The disc has alternating opaque and transparent sectors. The opaque sectors interrupt the light beam. A pulse is produced at the output of the light detector for each transparent sector. These pulses are counted to determine the speed.

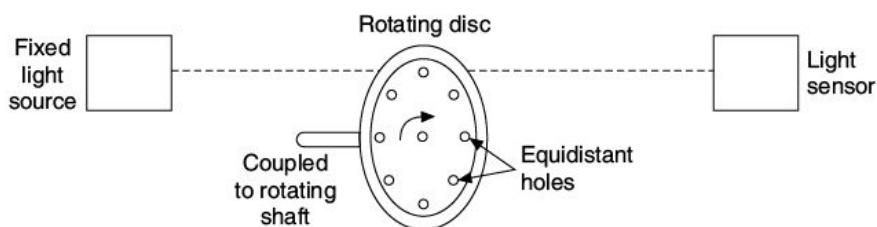


Figure 8.16 Photoelectric shaft speed sensing.

Generator output sensor

The current and voltage transformers are used to convert the voltage and current outputs of the generator into readable inputs to ADC. The ratings and types of transformers will depend on the generator outputs and the ADC circuit being used. The signal conditioning circuit may also be required.

Transducer mounting

A narrow tube is connected to the place where the steam pressure is required to be measured. The transducer is connected to the other end of the tube. The steam goes to the transducer and applies pressure, which is proportional to the pressure in the pipe. The LVDTs are mounted along with the valves as parts of the actuators. The speed detector is along with the ferromagnetic tooth wheel. The mounting of the photoelectric transducer may be tricky as it requires a light source and a light detector to be mounted in line, very accurately. The disc should be mounted parallel to these two.

The voltage and current transformers are connected to the generator output and are mounted along with the generator.

8.3.2 Multiplexer and Data Converter

Since there are nine analog sources of data, we would need a 16-channel analog multiplexer. It will be better to have a differential multiplexer for noise immunity. Since the multiplexers are available as 2, 4, 8, 16, ... channel, we should select one 16-channel differential multiplexer. Alternatively, two 16-channel multiplexers can be used in the differential mode. In the latter case, each channel of the upper multiplexer is connected to transducer output after signal conditioning whereas the corresponding channel of the lower multiplexer is connected to transducer ground voltage. The output of the speed detector is input to the timer/counter directly, in order to calculate the speed of the rotating shaft.

The choice of the analog-to-digital converter depends on the accuracy and the conversion time. Since the application does not require very fast processing, we may select any ADC with the conversion time in milliseconds. The 8, 10, 12, 16 bit ADCs are available commercially.

The sample-and-hold circuit will not be needed since the changes in pressure and speed are not very fast, and can be very well detected by the sampling rate in milliseconds.

8.3.3 Printer

The printer interface was described in Chapter 7. Let us assume that a simple Centronics printer is being used in the system for printing purposes. The print format should also be decided before writing the interface software.

8.3.4 Timer Function

The timer/counter chip is needed to generate the real time clock, so that the current time can also be printed along with the other parameters. The real time clock should give the time in hours, minutes and seconds and should be capable of working for 24 hours.

8.3.5 I/O Ports

The following input/output lines are required to interface the microprocessor to the printer, the ADC and the multiplexer in this application.

Multiplexer

Address lines for the multiplexer	4	Output
Analog-to-digital converter		
Start Convert line for ADC	1	Output
End of Conversion line from ADC	1	Input
Data lines from ADC (assuming an 8-bit ADC)	8	Input

Printer

Data lines	8	Output
Strobe line (\overline{SE})	1	Output
\overline{INIT} line	1	Output
BUSY line	1	Input
Acknowledge line (\overline{ACK})	1	Input
ERROR (\overline{ERROR}) Line	1	Input
Paper Exhaust line (PE)	1	Input

Thus, two 8-bit output ports and two 8-bit input ports will be needed to interface these devices to the microprocessor. Figure 8.17 shows the input ports 1 and 2 and the output ports 1 and 2 connected to the multiplexer, the ADC and the printer.

We shall develop both the 8085- and the 8086-based hardware designs and software for this application. Let us first design the hardware which will be common to both the 8085- and the 8086-based systems.

The hardware design shown in Figure 8.18 is the modification of the design in Figure 8.17. Since we need two input and two output ports, two 8255 chips are required. The input and output ports are implemented using two 8255 PPIs (8255-1 and 8255-2). The timer counters are implemented using the two 8253 timer/counter chips (8253-1 and 8253-2). These chips are selected using the address decoder 74LS138-1, which

itself is selected by the microprocessor through the chip select signal at the G1 pin. The memory interfacing is shown in a generalized way. The actual circuit will be different for the 8085 and the 8086 microprocessors.

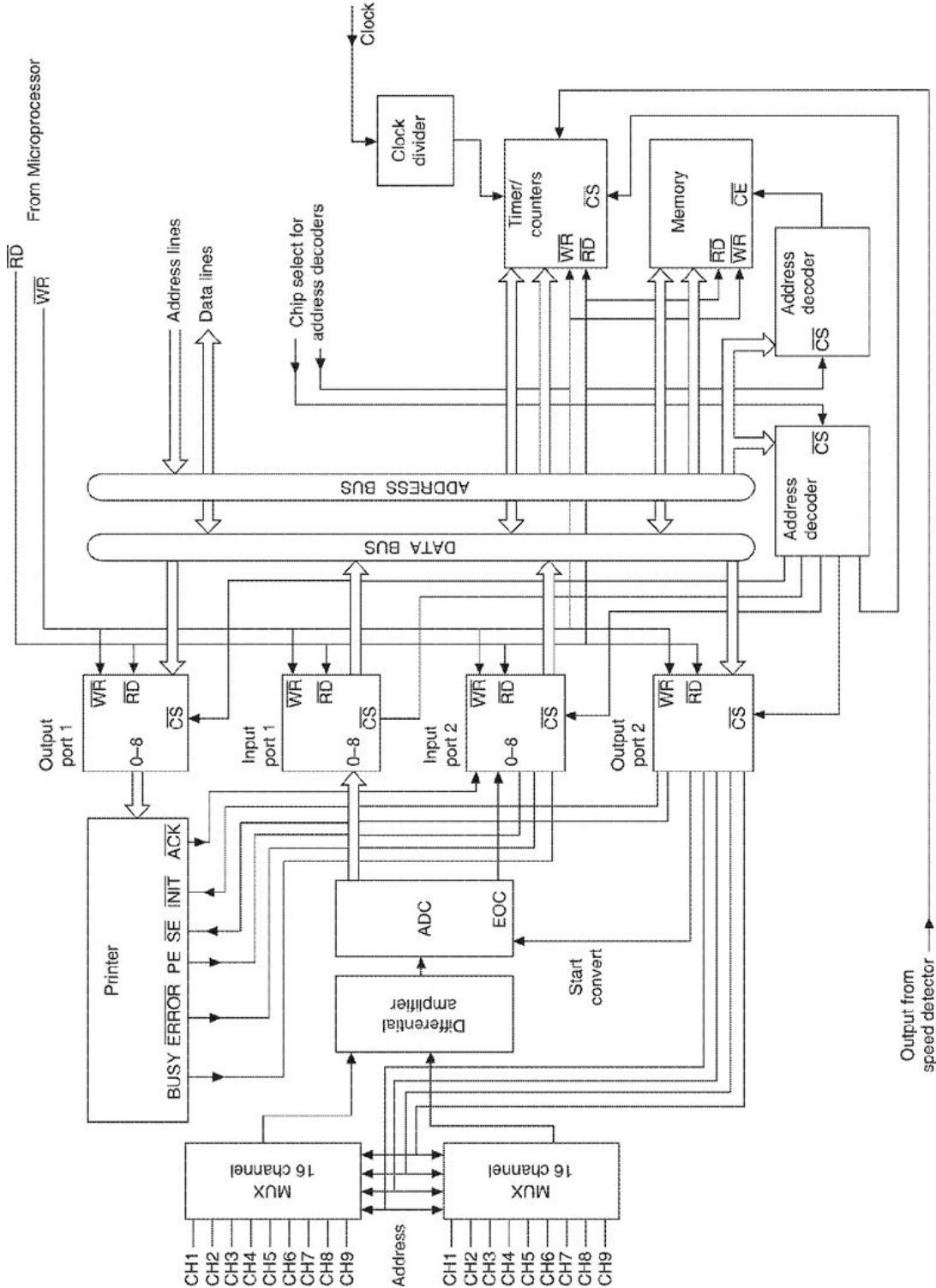


Figure 8.17 Block diagram of the turbine monitoring system.

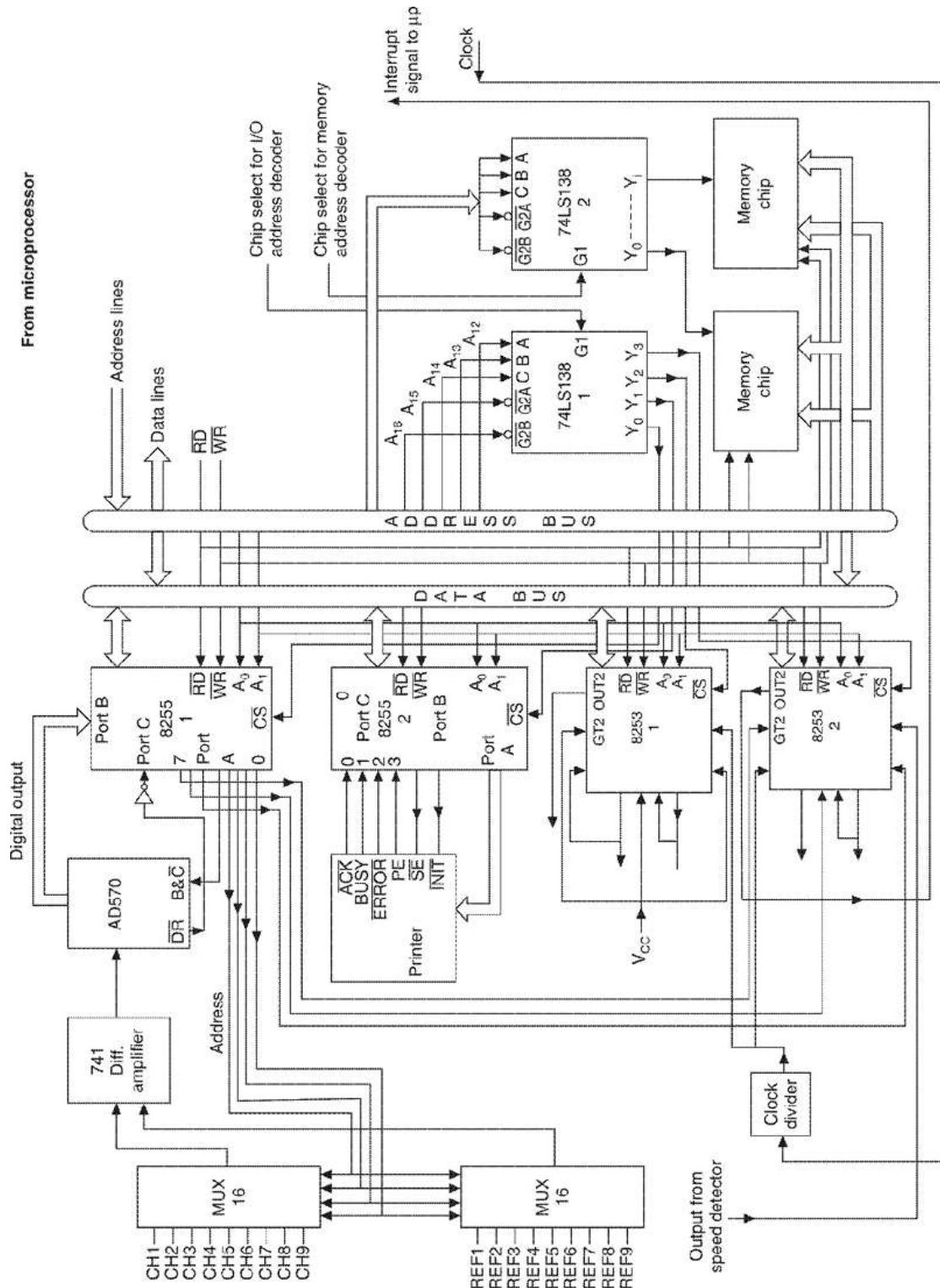


Figure 8.18 Turbine monitoring system with generalized microprocessor signals.

Since the 8253 accepts the clock signals which are less than 2 MHz frequency, the clock output of the microprocessor is divided through a clock divider. The extent to which the clock is divided, will depend on the particular microprocessor. The clock, after the division, is fed to counter 0 of the 8253-1 to obtain a 1 second pulse which is input to

counter 1 to get a 1 minute pulse. The 1 minute pulse is accepted by counter 2 to record time in minutes. This counter automatically resets to zero after 1440 minutes, i.e. 24 hours.

The analog portion consists of two analog multiplexers (MUX 16) and a 741 differential amplifier to get the differential output which is fed to AD570 ADC to get the corresponding digital value. Since the sampling frequency is low, the end-of-conversion pulse is not connected to the interrupt input of the microprocessor.

The other major components include the Centronics printer, interfaced using the 8255-2.

8.3.6 Software Flowchart

The software should perform the following tasks in sequence.

- (a) Initialization of the 8255 and the 8253
- (b) Read and store data from all the eight channels
- (c) Read the 8253 timer
- (d) Convert the values into engineering units
- (e) Compare with the last scanned values of the parameters
- (f) Print the current values with time if there is any change
- (g) Repeat from (b) to (f) continuously.

Figure 8.19 shows the flowchart.

The 8255 initialization

The 8255-1 is used to interface the multiplexer and the analog-to-digital converter. Port A is configured as the output port and is used to output the address information to the multiplexer and the Start Convert signal to the ADC. It is also used to output the initiation signal to gate inputs of the three counters of the 8253-2 chip. Port B is configured as input and connected to the ADC to read the digital data. Port C is configured as input. Bit 0 of port C is connected to the End-of-Conversion line of the ADC. The microprocessor reads the the End-of-Conversion signal status through Port C.

The control word, thus, will be 1 0 0 0 1 0 1 1 = 8BH.

The port numbers are 00H to 03H.

The 8255-2 is used to interface the printer to the microprocessor. The port configurations are as follows:

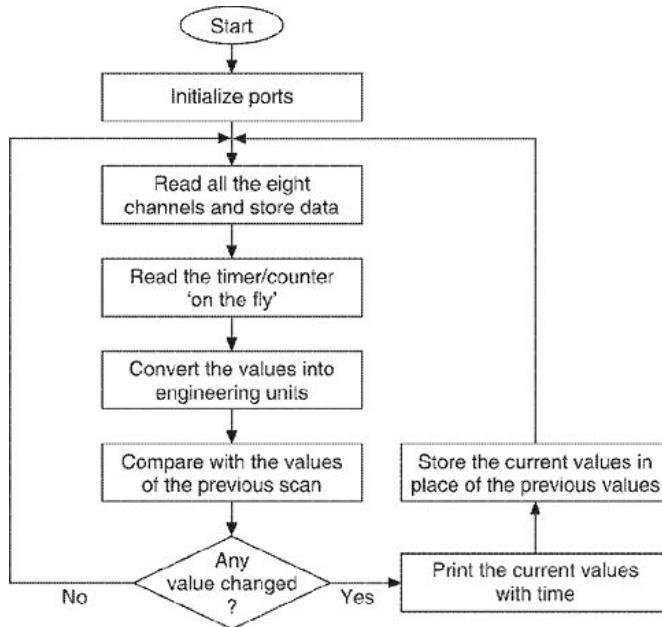


Figure 8.19 Flowchart for the turbine monitoring system.

Port A	Output	(Bit 0–Bit 7)	Data lines.
Port B <u>(INIT)</u> line.	Output	Bit 0	Initialization
		Bit 1	Strobe (\overline{SE})
Port 0 line.		(Bit 2–Bit 7)	Not used.
	C	Input	Bit
	Acknowledgement (\overline{ACK}) line.		
	Bit 1		Busy (BUSY)
	Bit 2		Error (\overline{ERROR})
	Bit 3		Paper Exhaust
(PE) line.			

The control word is $1\ 0\ 0\ 0\ 1\ 0\ 0\ 1 = 89H$.

The port numbers are 04H to 07H.

The 8253 initialization

The 8253-1 is used to generate the real time clock. The clock output of the microprocessor through the clock divider is fed to Counter 0 to obtain a 1 second pulse, which, in turn, is fed to counter 1 to obtain a 1 minute pulse. The 1 minute pulse is fed to Counter 2 to measure the time in minutes, whereas the contents of Counter 1 provide the seconds value

of time. The Counter 2 is initialized after 24 hours, i.e. 1440 minutes. All the counters are programmed in mode 2, i.e. the rate generator.

The Counter 0 is the binary counter, whereas the counters 1 and 2 are the BCD counters.

The port numbers are 08H to 0BH.

Counter 0

Control Word = 0 0 1 1 0 1 0 0 = 34H

The count value will depend on the frequency of the clock output of the microprocessor used.

Counter 1

Control Word = 0 1 0 1 0 1 0 1 = 55H

Since Counter 1 is used to generate the 1 minute pulse, the count value is 60, which is represented in BCD as 0 1 1 0 0 0 0 = 60H

Counter 2

Control Word = 1 0 1 1 0 1 0 1 = B5H

The Counter 2 should initialize itself after 1440 minutes. Thus, the count is 1440 which is represented in two bytes in BCD as follows

High-order Byte 0 0 0 1 0 1 0 0 = 14H

Low-order Byte 0 1 0 0 0 0 0 0 = 40H

The count is loaded with the low-order byte first, followed by the high-order byte.

The 8253-2 is employed to calculate the shaft speed. The output of the speed detector is fed to the clock input of Counter 0, which counts the number of pulses. The Counter 0 is used in conjunction with Counter 1. The output pulse of Counter 0 is fed to the clock input of Counter 1.

The clock pulse output from microprocessor after the clock divider is fed to Counter 2 of the 8253-2, which counts the number of these pulses to correlate the time with the number of pulses counted by Counter 1. The port numbers are 0CH to 0FH.

Counter 0

Control word = 0 0 1 1 0 1 0 0 = 34H

The mode set is 2 (i.e. the rate generator); it is a binary counter and the least significant byte is loaded first, followed by the most significant byte. The count loaded is FFFFH.

Counter 1

Control word = 0 1 1 1 0 0 0 0 = 70H

The count is same as in Counter 0. The mode set is 0, i.e. the interrupt on the terminal count. These two counters, together, can count up to $2^{32} - 1$, i.e. 4294967295 pulses. Assuming that there are 300 teeth in the wheel, the number of revolutions that can be counted are 14316557. With 3000 rpm speed of the shaft, the two counters can count up to 4772.18 minutes, i.e. more than three days.

Counter 2

Control word = 1 0 1 1 0 0 0 0 = B0H

The count and mode are same as Counter 1. The total number of counts, i.e. 65535 accounts for a few seconds, considering that the clock, after division, will have less than 0.1 millisecond period. The output of this counter is, therefore, connected to the microprocessor interrupt through which a software counter is maintained.

Conversion to engineering units

The conversion of digital values to engineering units will require scale and range of the ADC and the transducers. For the main steam pressure, the maximum value is 200 kg which is represented as 10 V. Similarly, the minimum value is 0 kg which is represented as 0 V. (The transducer output should be in the range 0–10 V. Thus the range of 200 kg is represented by the range of 10 V in the transducer, thereby giving 0.05 V for each kg increase in pressure.

Now, in ADC AD570, the range is 0–10 V for unipolar operation and 10 V is represented as FFH, i.e. 255. Thus,

$$\begin{array}{ccc} 200 & \text{kg} & 10 \\ \text{V} & = 255 & \\ & (\text{Steam pressure}) & (\text{Transducer output}) \\ & (\text{ADC output}) & \end{array}$$

Minimum measurable steam pressure, LSB in ADC = $200/255 = 0.78$ kg.

The engineering unit conversion program for the main steam pressure will thus take the digital output of ADC and multiply it with the conversion factor of 0.78 to get the absolute value of the main steam pressure in kg/cm. Similarly, the conversion of other values into engineering units will require multiplication with different conversion factors.

Printing

The printing of data on the printer will involve the transfer of the data from memory to the print buffer. Before transferring each line to the print buffer, the data should be arranged in ASCII code in the print format in memory. The print format, for our example, will include the time, the shaft speed, and the data from the transducers in the engineering units. It is also necessary to leave some blanks between two values while printing, in order to have better readability. A print format is shown in Figure 8.20.

```

TIME##XXHRS##YYMNTS##ZZSECSLF
MSP##RHSP##CON-VAC##GEN-CURR##VOLT##ILF
AAA##BBB##CCC##DDD##EEE##ELF
MSN-POS##CV-POS##ICV-POS##SHAFT-SPLF
FFF##GGG##HHH##III##LF
# = Blank
XX, YY, ZZ - Time in Hours Minutes and Second
AAA, BBB, CCC, DDD, EEE = Values of MSP, RHSP, CON-VAC, GEN-CURR and
GEN-VOLT respectively
FFF, GGG, HHH, III = Values of MSN-POS, CV-POS, ICV-POS and
SHAFT-SP respectively
LF = Line Feed character for going to next line

```

Figure 8.20 Print format.

Thus, in memory, 175 characters should be reserved for storing the data in the print format. The names and units like TIME, HRS, MNTS, SECS, etc. should be stored permanently and the data should be transferred when the printing is desired. The data should be converted to ASCII before storing in print memory. The conversion of binary data to ASCII will require that the data be first converted from binary to BCD and then to ASCII. After the print memory is full, the data is sent to the print buffer in printer, sequentially, character by character. When one line has been sent to the print buffer, the print command is sent. The same is repeated till all the lines are printed.

Now, let us introduce the microprocessor into our hardware design. We shall develop the hardware and the software using the 8085 and 8086 microprocessors, respectively.

8.3.7 Using the 8085

Figure 8.21 shows the hardware design of the turbine monitor using the 8085. It is assumed that a 6 MHz crystal is connected between X₁ and X₂ pins. Thus the CLK OUT frequency is 3 MHz. It is divided by 256

using the clock divider to get 11.71 kHz clock which is fed to counter 0 of the 8253-1 to obtain a 1 second pulse. The period of the clock is 0.085 millisecond. Thus, to obtain the 1 second pulse the count value of Counter 0 will be 11765 ($11765 \div 0.085 \text{ ms} = 1 \text{ second}$). It will be represented as 2DF5H in hexadecimal.

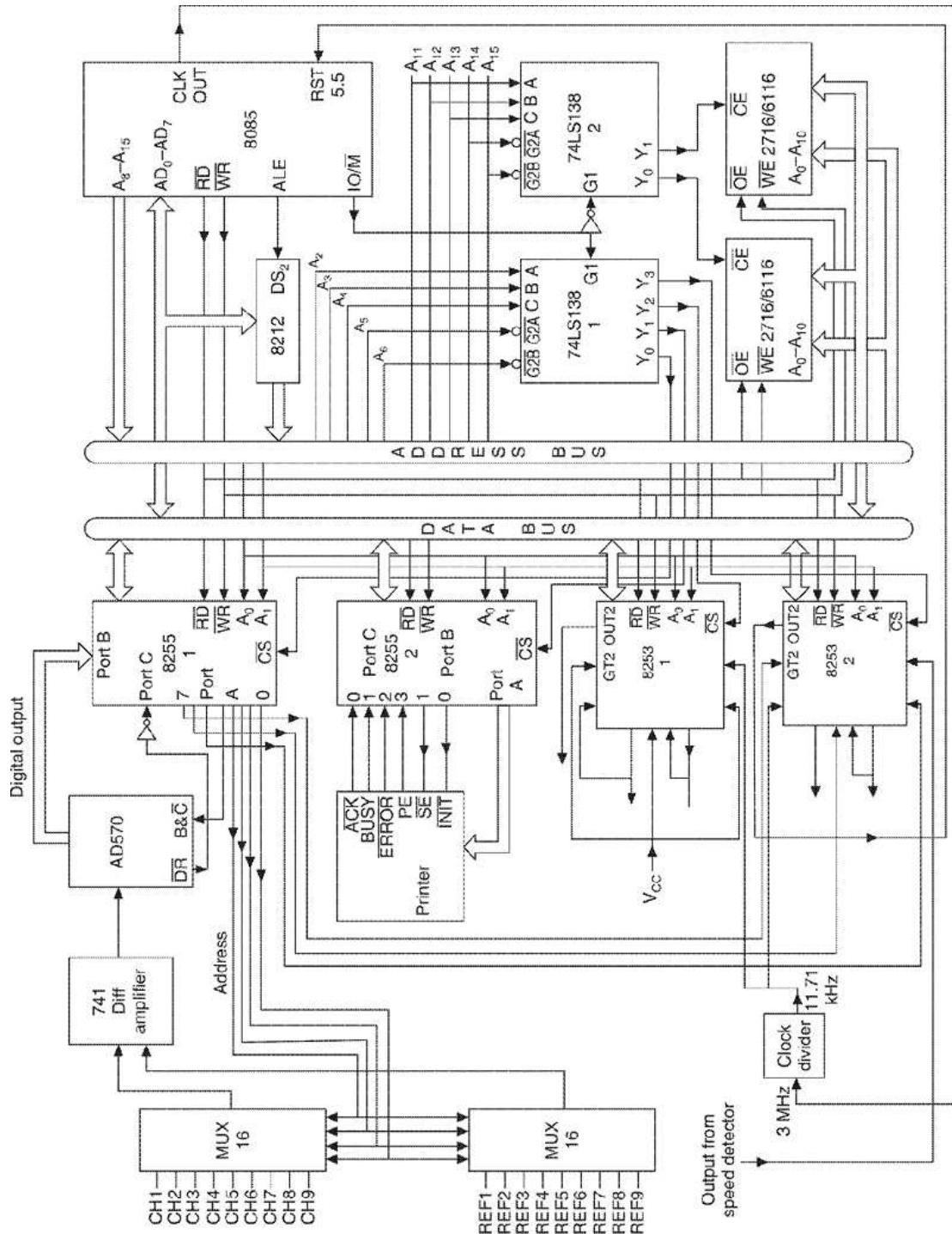


Figure 8.21 Turbine monitoring system using the 8085.

The 0.085 ms pulse from the clock divider is also input to Counter 2 of Intel 8253-2, which counts the number of these pulses to correlate the time with the number of pulses counted by counter 1 of the 8253-2 (for the purpose of shaft speed calculation).

The output of Counter 2 is connected to RST 5.5 interrupt input. The Interrupt Servicing Routine of RST 5.5 implements the software counter. Since the total number of counts 65535 (of Counter 2) accounts for 0.085 ms $\square 65535 = 5.57$ seconds, the software counter is necessary. Each count of the software counter is equal to 5.57 seconds.

Two memory chips of 2 KB each (2716/6116) have been interfaced. These chips are selected through the address decoder 74LS138-2 which itself is selected through the IO/M signal at G1 pin.

We shall now attempt to develop the program, which is as follows:

```
;
; Intel 8085 program for monitoring of turbine parameters
;
    DATA      DSB   9
    COUNT    DSB   1
    FLAG     DSB   1
    FACT     DSB   9
    DATAP    DSB   9
    PMEM     DSB  200
    CN1-CNT0L DSB   1
    CN1-CNT0H DSB   1
    CN1-CNT1L DSB   1
    CN1-CNT2L DSB   1
    CN1-CNT2H DSB   1
    CN2-CNT0L DSB   1
    CN2-CNT0H DSB   1
    CN2-CNT1L DSB   1
    CN2-CNT1H DSH   1
    CN2-CNT2L DSB   1
    CN2-CNT2H DSB   1

; Main Program
    START:   CALL  INIT
    LOOP0:   CALL  SCAN; Read all the 9 channels and store
              data
              CALL  RCNT; Read the contents of Intel 8253 counters
              and store
```

CALL CONV; Convert the data into engineering values.
 CALL COMP; Compare the present data with the past scan data
 LDA FLAG; FLAG = 1, if change in data, otherwise FLAG = 0
 ANA A; Load and check flag.
 JZ LOOP0
 CALL PRINT; Print new values
 CALL STORE; Store new values in place of old
 JMP LOOP0

;

; Subroutine SCAN

;

SCAN: MVI C, 09H; C Register used as counter
 MVI B, 00H; B Register contains the channel number
 LXI H, DATA; Address of the data stored in H-L register

; Initiate MUX through Port A. B& \bar{C} pin of AD 570 = high. GT0, GT1, GT2 pins of ; 8253-2 = high.

SCN1: MOV A, B; (A) = Channel No.
 ORI F0H
 OUT 00H;

; Send Start Convert signal. B& \bar{C} pin of AD 570 = low. GT0, GT1, GT2 pins of ; 8253-2 = high. MUX address lines = Channel No.

MOV A, B
 ORI E0H
 OUT 00H

SCN5: IN 02H; Read End-of-Conversion from port C
 CPI 01H; Check if EOC = 1
 JNZ SCN5
 IN 01H; Read the digital output from port B
 MOV M, A; Store data
 INX H; Increment memory address
 INR B; Increment the channel number
 DCR C; Decrement the count
 JNZ SCN1; Repeat the scan if all channels

; are not over otherwise return
RET

;

; Subroutine INIT

;

INIT:	MVI A, 8BH; initialize 8255-1
OUT	03H
MVI	A, 89H; initialize 8255-2
OUT	07H
MVI	A, 34H; initialize 8253-1 Counter 0
OUT	0BH
MVI	A, 55H; initialize 8253-1 Counter 1
OUT	0BH
MVI	A, B5H; initialize 8253-1 Counter 2
OUT	0BH
MVI	A, 34H; initialize 8253-2 Counter 0
OUT	0FH
MVI	A, 70H; Initialize 8253-2 Counter 1
OUT	0FH
MVI	A, B0H; Initialize 8253-2 Counter 2
OUT	0FH
MVI	A, F5H; Load 8253-1 Counter 0, the lower byte
OUT	08H
MVI	A, 2DH; Load 8253-1 Counter 0, the higher
byte	
OUT	08H
MVI	A, 60H; Load 8253-1, Counter 1, byte
OUT	09H
MVI	A, 40H ; Load 8253-1, Counter 2, lower byte
	; and higher byte
OUT	0AH
MVI	A, 14H
OUT	0AH
MVI	A, FFH; Load 8253-2 Counter 0
OUT	0CH
OUT	0CH
OUT	0DH; Load 8253-2 Counter 1
OUT	0DH
OUT	0EH; Load 8253-2, Counter 2
OUT	0EH

; Initiate 8253-2 counters by making GT0, GT1 and GT2 = high. B& \bar{C}
 pin of
 ; AD 570 = high.
 MVI A, F0H
 OUT 02H
 RET
 ;
 ; Subroutine RCNT
 ;
 ; Inhibit counting in counters of 8253-2 by making GT0, GT1 and
 GT2 = low
 ; B& \bar{C} pin of AD 570 = high.
 RCNT: MVI A, 10H
 OUT 00H
 IN 0CH
 STA CN2-CNT0L
 IN 0CH
 STA CN2-CNT0H
 IN 0DH
 STA CN2-CNT1L
 IN 0DH
 STA CN2-CNT1H; 32 bit count for speed detector
 ; pulses in CN2-CNT1H to CN2-CNT0L
 IN 0EH
 STA CN2-CNT2L
 IN 0EH
 STA CN2-CNT2H
 ; No. of 0.085 ms pulses in CN2-CNT2H and CN2-CNT2L
 ;
 ; Load the count from memory. Count software counter associated
 with counter through
 ; RST 5.5. interrupt.
 ;
 LDA COUNT
 CALL SCALC; Calculate the speed in rpm and store
 MVI A, 00H; Read Counter 0 of 8253-1 'on the fly'
 OUT 0BH
 IN 08H
 STA CN1-CNT0L
 IN 08H

```

STA      CN1-CNT0H ; Counter 0 counts in CN1-
CNT0H and
; CN1-CNT0L
MVI      A, 40H; Read Counter 1 of 8253-1 'on the fly'
OUT     0BH
IN      09H
STA      CN1-CNT1L; Counter 1 counts in CN1-
CNT1L
MVI      A, 80H; Read Counter 2 of 8253-1 'on the fly'
OUT     0BH
IN      0AH
STA      CN1-CNT2L
IN      0AH
STA      CN1-CNT2H ; Counter 2 counts in CN1-
CNT2H and
; CN1-CNT2L
CALL    TCALC; Calculate the time and store
RET

;
;

; Subroutine CONV
;
CONV:           LXI      H, DATA; address of data in H-L
 registers
LXI      D, FACT; conversion factors for various channels are
; stored sequentially from address FACT
MVI      C, 09H
;
; Multiplication routine multiplies digital data from ADC, stored at
DATA and the
; conversion factor stored at FACT. The result is stored in DATA.
;
CONV1:          CALL    MULT
INX      H
INX      D
DCR      C
JNZ      CONV1
RET

;
; Subroutine COMP

```

```

;

COMP:      MVI   A,00H
           STA   FLAG; Make FLAG = 0
           LXI   H, DATA; Present the scan data address
           LXI   D, DATAP; Previous scan data address
           MVI   C, 09H; Register C used as counter
           LDAX  D

COMP1:     CMP   M; Compare
           JNZ   COMP2; Jump if unequal
           INX   D; Compare further
           INX   H
           DCR   C
           JNZ   COMP1
           RET

COMP2:     MVI   A, 01H; Make FLAG = 1
           STA   FLAG
           RET

;

;

;

; Subroutine PRINT
;

PRINT:    MVI   A, 0EH; Initialize printer by sending
          the INIT signal
          OUT  05H
          LXI  H, PMEM; PMEM is the starting address of the
          print ; memory
          LXI  D, DATA
          CALL CHNGE ; Change the data into ASCII and store
          in    ; PMEM
          MVI  D, 05H; Counter for 5 lines
          LXI  H, PMEM

;

;

; Output the character code to print buffer through port A of 8255-2.
;

CPRNT:   MOV   A, M
          MOV   B, A; Store character in register B
          OUT  04H
          MVI  A, 0DH; Make SE = 0

```

```

        OUT    05H
WAIT:      IN     06H; Read ACK, check if ACK = 0
        ANI    01H
        JNZ    WAIT
        MVI    A, 0FH; Make SE=1
        OUT    05H
        INX    H
        IN     06H
        MOV    E, A; (E) = (A)
CHBSY:    ANI    02H; Check for busy
        JNZ    CHBSY
CHEROR:   MOV    A, E; Check for error
        ANI    04H
        JZ     CHEROR
CHPE:     MOV    A, E; Check for paper exhaust
        ANI    08H
        JNZ    CHPE
        MOV    A, B; Check if the previous character was LF
        CPI    0AH
        JNZ    CPRNT
        CALL   DELAY; Delay for 4 seconds
        DCR    D; Decrement the line counter
        JNZ    CPRNT
        RET

;
;

; Subroutine STORE
;
STORE:    LXI    H, DATA
        LXI    D, DATAP
        MVI    C, 09H; Counter for the number of data bytes.
STOR1:   MOV    A, M
        STAX   D
        INX    H
        INX    D
        DCR    C
        JNZ    STOR1
        RET
;
```

; RST 5.5 interrupt servicing routine for resetting of Counter 2 and increment COUNT.

;

RST 5.5: LDA COUNT

 INR A

 STA COUNT

 MVI A, FFH

 OUT 0EH

 OUT 0EH

;

; Initiate through 8255-1 port-A bit 7

;

 MVI A, 80H

 OUT 00H

 RET

; Subroutine SCALC

SCALC:

—

—

—

RET

; Subroutine TALC

TALC:

—

—

—

RET

; Subroutine MULT

MULT:

—

—

—

RET

; Subroutine CHNGE

CHNGE:

—

—

—

RET

; Subroutine DELAY

DELAY:

—
—
—

RET
END

8.3.8 Using the 8086

The hardware design of the turbine monitor using the 8086 is shown in Figure 8.22. It is assumed that a 6 MHz crystal is connected between X₁ and X₂ pins. The frequency at PCLK will be 3 MHz which is divided by 256, using the clock divider to get 11.71 kHz clock (with time period = 0.085 ms). It is connected to Counter 0 of the 8253-1 to get the 1 second pulse.

The period of the clock is 0.085 ms. Thus, to obtain the 1 second pulse, the count value of Counter 0 = 1 second ÷ 0.085 ms = 11765. It is represented as 2DF5H in hexadecimal.

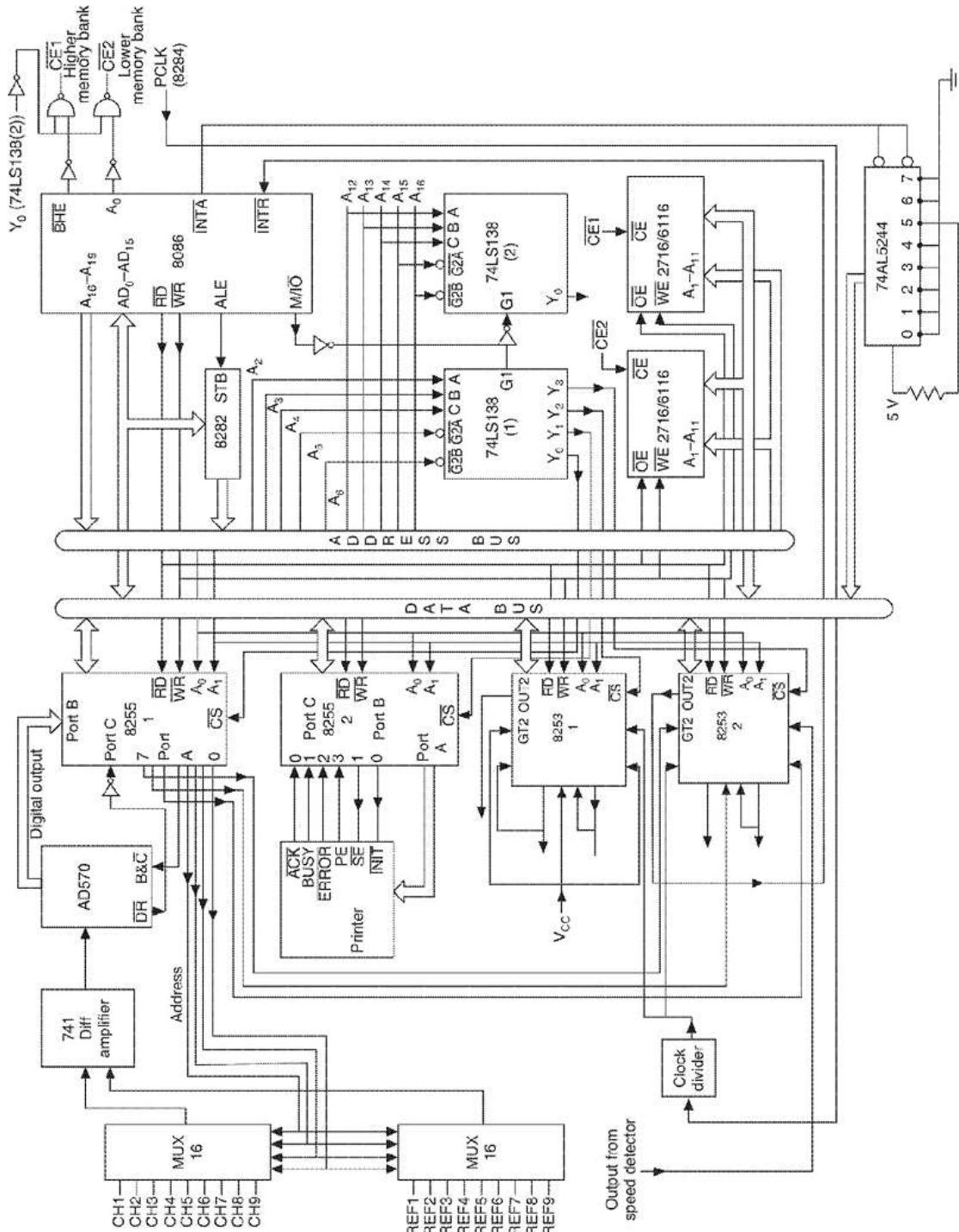


Figure 8.22 Turbine monitoring system using the 8086.

The 0.085 ms pulse output from the clock divider is also input to Counter 2 of the 8253-2 which counts the number of these pulses to correlate the time with the number of pulses (from shaft encoder) counted by Counter 1 of the 8253-2. The purpose is to calculate the shaft speed. The output of Counter 2 is connected to the INT interrupt input. The interrupt acknowledgement INTA causes the loading of vector type 20H to the data

bus. Thus, type 20H, i.e. type 32 interrupt is caused. The interrupt servicing routine implements the software counter.

Since the total number of counts 65532 (of Counter 2) accounts for $0.085 \text{ ms} \times 65532 = 5.57 \text{ seconds}$, the software counter becomes necessary. Each count of the software counter is equal to 5.57 seconds. We shall now attempt to develop the program, which is as follows:

```
;
; Intel 8086 program for the turbine monitor
;
; Main program
;

        DATA SEGMENT PUBLIC
        DATA      DSB      9
        COUNT    DSB      1
        FLAG     DSB      1
        FACT     DSB      9
        PMEM    DSB     200
        DATA-CN  DSW      9
        DATA-CNP DSW      9
        PL-CNT0  DSB      1
        PL-CNT1  DSB      1
        RPM      DSB      1
        CNT0    DSB      1
        CNT1    DSB      1
        CNT2    DSB      1
        CNT3    DSB      1
        TCNT00  DSB      1
        TCNT01  DSB      1
        TCNT10  DSB      1
        TCNT20  DSB      1
        TCNT21  DSB      1
        HRS     DSB      1
        MINT    DSB      1
        SECND   DSB      1

        DATA ENDS

        S-STACK SEGMENT
                DW 100 DUP (0)
                T-STACK LABEL WORD
        S-STACK ENDS

;
```

```

; Declare ISR CNTISR2 in other module
; PROCEDURES SEGMENT PUBLIC
    EXTRN CNTIRS2: FAR
PROCEDURES ENDS
    PUBLIC COUNT
CODE SEGMENT PUBLIC
    ASSUME CS:CODE, DS:DATA: SS: S-STACK
;
; Initialize the segment registers.
;
START:    MOV      AX, DATA
          MOV      DS, AX
          MOV      AX, S-STACK
          MOV      SS, AX
          MOV      SP, OFFSET T-STACK
;
; Insert the address of ISR CNTIRS2 in interrupt vector table at
0080H to 0083H.
;
          MOV      AX, 0000H
          MOV      ES, AX
          MOV      WORD PTR ES: 0080H, OFFSET
          CNTIRS2
          MOV      WORD PTR ES: 0082H, SEG CNTIRS2
;
; Initialize the 8255 and the 8253 timer/counter chips.
;
          CALL    INIT
;
; Read all the nine channels and store data.
;
LOOP0:    CALL    SCAN
          CALL    RCNT
;
; Convert the data into engineering values.
;
          CALL    CONV
;
; Compare the present data from the past scan data. FLAG = 1, if
change in data
; otherwise FLAG = 0.

```

```
;  
    CALL      COMP  
;  
; Load and check FLAG.  
;  
    MOV      AL, FLAG  
    AND      AL, FFH  
    JZ       LOOP0  
;  
; Change in values. Print new values.  
;  
    CALL      PRINT  
; Store new values.  
;  
    CALL      STORE  
    JMP      LOOP0  
;  
; Procedure INIT to initialize the 8253 and the 8255 chips.  
    INIT    PROC    NEAR  
;  
; Initialize the 8255 chips.  
;  
    MOV      AL, 8BH; Initialize 8255-1  
    OUT     03H, AL  
    MOV      AL, 89H; Initialize 8255-2  
    OUT     07H, AL  
;  
; Initialize the 8253 chips.  
;  
    MOV      AL, 34H; Initialize 8253-1 Counter 0  
    OUT     0BH, AL  
    MOV      AL, 55H; Initialize 8253-1 Counter 1  
    OUT     0BH, AL  
    MOV      AL, B5H; Initialize 8253-1 Counter 2  
    OUT     0BH, AL  
    MOV      AL, 34H; Initialize 8253-2 Counter 0  
    OUT     0FH, AL  
    MOV      AL, 70H; Initialize 8253-2 Counter 1  
    OUT     0FH, AL  
    MOV      AL, B0H; Initialize 8253-2 Counter 2
```

```

        OUT      0FH, AL
;
; Load counts in various counters.
;
        MOV      AL, F5H ; Load the lower byte in Counter
0
; of 8253-1
        OUT      08H, AL
        MOV      AL, 2DH ; Load the higher byte in
Counter 0
; of 8253-1
        OUT      08H, AL
        MOV      AL, 60H ; Load 1 byte count in Counter 1
; of 8253-1
        OUT      09H, AL
        MOV      AL, 40H ; Load the lower byte in Counter
2
; of 8253-1
        OUT      0AH, AL
        MOV      AL, 14H ; Load the higher byte in Counter
2
; of 8253-1
        OUT      0AH, AL
        MOV      AL, FFH ; Load the lower and the higher
; bytes in Counter 0 of 8253-2
        OUT      0CH, AL
        OUT      0CH, AL
        OUT      0DH, AL ; Load the lower and the higher
; bytes in Counter 1 of 8253-2
        OUT      0DH, AL
        OUT      0EH, AL ; Load the lower and the higher
; bytes in Counter 2 of 8253-2
        OUT      0EH, AL
; Initiate the 8253-2 counters by making GT0, GT1 and GT2 = high.
B&C pin of AD
; 570 = high.
        MOV      AL, F0H
        OUT      02H, AL
        RET
INIT ENDP

```

```

;

; Procedure to scan channels and store data
;

SCAN PROC NEAR
    MOV CX, 09H; Counter
    MOV DL, 00; Channel No.
    LEA BX, DATA

; Initiate MUX through Port A. B&C pin of AD 570 = high. GT0, GT1
and GT2 of
; 8253-2 = high.

SCN1:    MOV AL, DL; (AL) = Channel No.
        OR AL, F0H
        OUT 00H, AL

; Send Start Convert Signal B&C pin of AD 570 = low. GT0, GT1 and
GT2 of
; 8253-2 = high. MUX address lines = Channel No.

        MOV AL, DL
        OR AL, E0H
        OUT 00H, AL

SCN5:    IN AL, 02H; Read End of
Conversion (EOC) from port C
        CMP AL, 01H; Check if EOC = 1
        JNZ SCN5
        IN AL, 01H; Read digital output from port B
        MOV (BX), AL; Store data
        INC BX
        INC DL
        LOOP SCN1
        NOP
        RET
        SCAN ENDP

;

; Procedure to read the contents of Intel 8253 counters and store
;

RCNT PROC NEAR
;

; Inhibit counting in counters of 8253-2 by making GT0, GT1 and
GT2 = low.
; B&C pin of AD 570 = low
;

```

```

        MOV      AL, 10H
        OUT      00H, AL
;
; Read Counter 0 and Counter 1 of Intel 8253-2 to find the number of
pulses of shaft
; encoder.
        IN       AL, 0CH
        MOV     CNT0, AL
        IN       AL, 0CH
        MOV     CNT1, AL
        IN       AL, 0DH
        MOV     CNT2, AL
        IN       AL, 0DH
        MOV     CNT3, AL
;
; The 32 bit count of speed detector is in CNT0 to CNT3. Read the
number of 0.085 ms
; pulses from Counter 02 of the 8253-2.
;
        IN       AL, 0EH
        MOV     PL-CNT0, AL
        IN       AL, 0EH
        MOV     PL-CNT1, AL
;
; COUNT variable contains the count of the software counter, which
stores the number of
; times the counter 2 has overflowed. The total time taken for speed
detector count (stored
; in CNT0 to CNT3) = [COUNT □ 65532 + (FFFFH – PL-CNT)] □
0.085 ms. The rpm
; of the speed detector = total count/total time in minutes.
;
        CALL    SCALC
;
; Procedure SCALC takes PL-CNT0, PL-CNT1, COUNT and CNT0
to CNT3 input and
; calculates the rpm speed of the speed detector shaft, and stores as
RPM.

```

; RPM = total count (represented by CNT0 to CNT3 in 32 bits)/total time in minutes
 ; (represented by PL-CNT and COUNT)
 ;
 ; Calculate the time. To calculate the time, read 8253-1 counters on the fly, i.e.
 ; without inhibiting counting.
 ;
 MOV AL, 00H; Send command to 8253-1 to
 ; read Counter 0 on the fly
 OUT 0BH, AL
 IN AL, 08H
 MOV TCNT00, AL
 IN AL, 08H
 MOV TNCT01, AL
 ;
 ; Counter 0 count in TCNT00 & TCNT01
 ;
 MOV AL, 40H; Send command to read Counter
 1 on the fly
 OUT 0BH, AL
 IN AL, 09H
 MOV TCNT10, AL; (AL) □ Counter 1 count
 ;
 ; Counter 1 count in TCNT10
 ;
 MOV AL, 80H; Send command to read Counter
 2 on the fly
 OUT 0BH, AL
 IN AL, 0AH
 MOV TCNT20, AL
 IN AL, 0AH
 MOV TCNT21, AL
 ;
 ; Counter 2 counts in TCNT20 and TCNT21
 ;
 CALL TCALC
 ;
 ; Procedure TCALC calculates the time in hours, minutes and seconds based on the values

```

; of TCNT00, TCNT01, TCNT10, TCNT20 and TCNT21 and stores
these in HRS, MINT
; and SECND
;
RET
RCNT    ENDP

CONV    PROC      NEAR
;
; Procedure to convert data on different channels in engineering units
; Conversion factors for various channels are stored in the array
named FACT
;
LEA        BX, DATA
LEA        BP, DATA-CN
LEA        DI, FACT
MOV        CX, 09H
CONV1:   MOV        AL, (DI)
          MUL        BYTE PTR (BX)
;
; AX contains the multiplication of FACT and DATA value for a
channel. Store the result
; in DATA-CN, i.e. data after conversion to engineering units.
;
MOV        (BP), AX
INC        BX
INC        BP
INC        DI
LOOP      CONV1
NOP
RET
CONV    ENDP
COMP    PROC      NEAR
;
; This procedure compares the data of the current scan with the data of
the past scan. If
; the two data are equal then FLAG = 0 otherwise FLAG = 1.
;
MOV        AL, 00H
MOV        FLAG, AL

```

```

;

; DATA-CN contains the present scan data and DATA-CNP contains
the previous scan
; data.

        LEA      BX, DATA-CN
        LEA      BP, DATA-CNP
        MOV      CX, 09H
COMP1:   MOV      AX, (BX)
        CMP      AX, (BP); Compare
        JNZ      COMP2; Jump if not equal
        INC      BP
        INC      BX
        LOOP
        JMP      COMP1
FINI:    MOV      AL, 01H
        MOV      FLAG, AL
FIN1:    NOP
        RET
COMP     ENDP

PRINT   PROC      NEAR
;

;

        CALL      CHNGE
;

; CHNGE procedure takes the data from HRS, MINT, SECND and
DATA-CN array,
; changes it into ASCII and stores in PMEM. The format of PMEM is
shown
; in Figure 8.20.
;
; Initialize printer through port B of the 8255-2.
; SE is connected to bit 1 and INIT is connected to bit 0 of port B of
8255-2.
;
        MOV      AL, 0EH
        OUT      05H, AL
        LEA      BX, PMEM
        MOV      CX, 05H; Counter for 5 lines
CPRNT:  MOV      AL, (BX)

```

```

    MOV      BL, AL ; Store character in BL register
    OUT      04H, AL ; through port A of the 8255-2
    MOV      AL, 0DH; Send load pulse ( $\overline{SE}$ )
    OUT      05H, AL

;

; Check for  $\overline{ACK}$ .
;

WAIT:     IN       AL, 06H
          AND      AL, 01H
          JNZ      WAIT
          MOV      AL, 0FH; MAKE  $\overline{SE} = 1$ 
          OUT      05H, AL
          INC      BX
          IN       AL, 06H
          MOV      DL, AL; (DL) = (AL)

;

; Check for busy.
;

CHBSY:    AND      AL, 02H
          JNZ      CHBSY

;

; Check for error.
;

CHEROR:   MOV      AL, DL
          AND      AL, 04H
          JZ       CHEROR

;

; Check for Paper Exhaust.
;

CHPE:     MOV      AL, DL
          AND      AL, 08H
          JNZ      CHPE

;

; Check if the previous character was LF?
;

          MOV      AL, BL
          CMP      AL, 0AH
          JNZ      CPRNT

;

; Introduce a delay of 4 seconds for a line to print.
;
```

```
;  
    CALL      DELAY  
    LOOP      CPRNT  
    NOP  
    RET  
PRINT  ENDP  
STORE  PROC      NEAR  
;  
; Procedure to store new values in place of the values previously  
stored in array  
; DATA-CNP . New values stored in array DATA-CN.  
;  
    LEA      DI, DATA-CNP  
    LEA      SI, DATA-CN  
    MOV      CX, 09H  
STOR1:   MOV      AX, (SI)  
          MOV      (DI), AX  
          INC      DI  
          INC      SI  
          LOOP     STOR1  
          NOP  
STORE  ENDP  
  
SCALC  PROC      NEAR  
:  
:  
RET  
SCALC  ENDP  
  
TALC   PROC      NEAR  
:  
:  
RET  
TALC   ENDP  
  
CHANGE PROC      NEAR  
:  
:  
RET  
CHANGE ENDP  
  
DELAY  PROC      NEAR
```

```

        :
        :
        RET
DELAY    ENDP
CODE     ENDS

DATA     SEGMENT      PUBLIC
        EXTRN      COUNT
DATA     ENDS
        PUBLIC      CNTIRS2
PROCEDURES SEGMENT      PUBLIC
        CNTIRS2 PROC      FAR
        ASSUME CS: PROCEDURES, DS:DATA
;
; Increment COUNT
;
        INC       COUNT
;
; Reset counter 2 of 8253-2 and load FFFFH
;
        MOV       AL, FFH
        OUT      0EH, AL; Lower byte loaded
        OUT      0EH, AL; Higher byte loaded
;
; Initiate through 8255-1 port A bit 7
;
        MOV       AL, 80H
        OUT      00H, AL
        IRET
CNTIRS2 ENDP
PROCEDURES ENDS

```

8.4 CONCLUSION

The two case studies dealt with in this chapter clearly illustrate the system design concepts using the 8085 and the 8086 microprocessors. The strengths and weaknesses of these microprocessors are also evident in these two case studies. These will become more clear as we study advanced processors in the subsequent chapters.

EXERCISES

1. Develop the remaining subroutines/procedures of Case Study 2 and complete the project. Test the system in a simulated environment.
2. The design of Case Study 2 envisages sequential channel scanning and printing after one cycle of the channel scanning is completed. During that time, no channel is scanned. Printing is a slow process and this may cause some alarm to be missed. This situation will not be acceptable. Modify the system design and the software
 - (a) to have channel scanning after every second caused through a timer interrupt, and
 - (b) to have printing when no other processing is taking place.
3. Incorporate a 4 × 4 key keyboard interface in the design and develop the software to
 - (a) input initial data in real time clock.
 - (b) input limits for various channels.

FURTHER READING

- Douglas, V. Hall, *Microprocessors and Interfacing Programming and Hardware*, Tata McGraw-Hill, 1992.
- Krishna Kant, *Computer Based Industrial Control*, Prentice-Hall of India, 1997.
- Krishna Kant, *Microprocessor Based Data Acquisition System Design*, Tata McGraw-Hill, 1987.
- Miller, Richard K. and Walker, Terri C., *Artificial Intelligence Applications in Sensors and Instrumentations*, SEAI Technical Publications, 1988.
- Norton, Harry N., *Handbook of Transducers*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

9

INTEL 8051 MICROCONTROLLER HARDWARE ARCHITECTURE

9.1 INTRODUCTION

So far we have dealt with the microprocessor architecture which is simple in nature. The 8085 and the 8086 microprocessors are industry standard and are quite versatile. We have dealt with their hardware architectures, facilities, programming as well as interfacing. We have also dealt with some real-life applications using these microprocessors. However, it must be borne in mind that these processors are not specifically designed for real-time applications. In fact, they have a number of limitations for real-time applications, because of which, more and more advanced microprocessors have been developed and are being used. We have covered these developments in Chapter 2 in brief. We have also discussed some real-time applications in Chapter 8.

In the present chapter, we will introduce the microcontroller Intel 8051, which has been widely used for embedded system applications. We will cover the facilities offered by the 8051 microcontroller and its hardware architecture. The instruction set and programming would be covered in the next chapter.

The 8051 is a stand-alone high performance microcontroller intended for use in sophisticated real-time applications, such as instrumentation, industrial control, automobiles and computer peripherals. It provides extra features like interrupts, bit addressability, and an enhanced set of instructions which make the chip very powerful and cost effective.

The 8051 may be used as a controller for many applications that require up to 64 KB of program memory and/or 64 KB of data memory. The ease with which it can be interfaced with the other peripheral chips, makes it quite versatile.

9.2 ARCHITECTURE

The block diagram of the 8051 is shown in Figure 9.1.

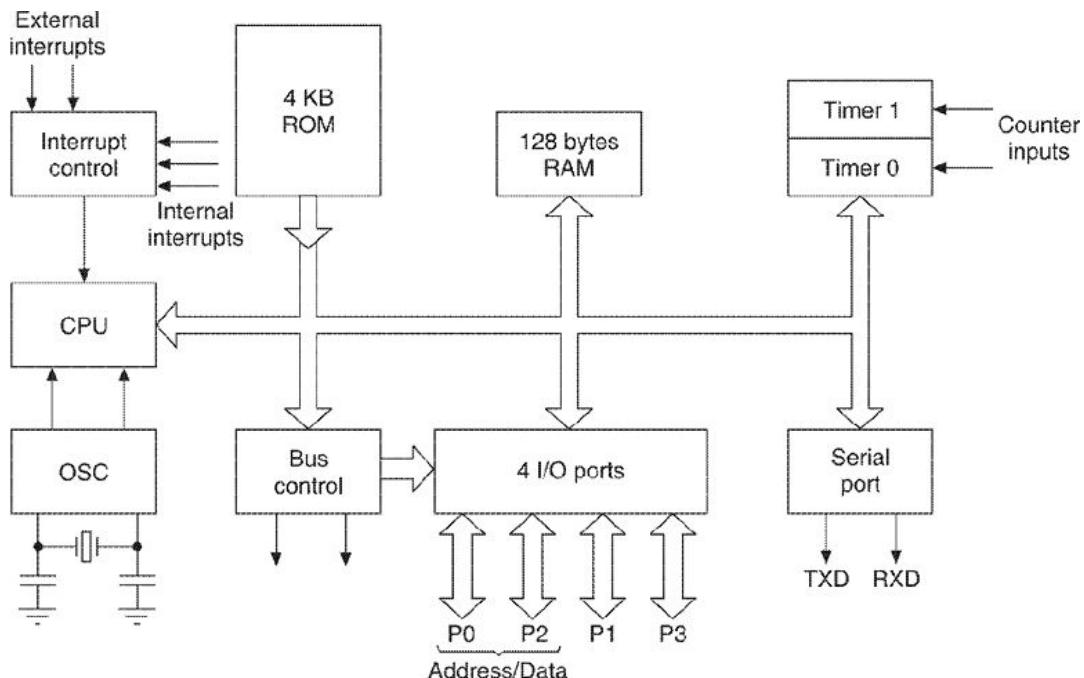


Figure 9.1 Block diagram of the 8051.

Following are the resources offered by the 8051 microcontroller:

1. 8-bit CPU
2. On-chip oscillator
3. 4 KB of ROM (Program memory)
4. 128 bytes of RAM (Data memory)
5. 21 Special function registers
6. 32 I/O lines (Ports P0 to P3)
7. 64 KB address space for external data memory
8. 64 KB address space for program memory
9. Two 16-bit timer/counter
10. Five source interrupt structure
11. Full duplex serial port
12. Bit addressability
13. Powerful bit processing capability

The MCS-51 family includes pin compatible chips like:

- Intel 8031
- Intel 8051

- Intel 8751

The 8031 is the ROM-less version of the 8051 microcontroller whereas the 8751 is the EPROM version of the 8051 microcontroller. It means that in case of the 8031, external ROM needs to be interfaced, whereas in case of the 8751, the on-chip 4 KB EPROM can be programmed for any application.

Some additional features have been provided in some of the devices in MCS-51 family. Table 9.1 shows the different devices along with their features.

Table 9.1 The MCS-51 family of microcontrollers

Device name	ROM-less version	EPROM version	ROM bytes	RAM bytes	16-bit timers	Ckt type
Intel 8051	Intel 8031	Intel 8751	4K	128	2	HMOS
Intel 8051AH	Intel 8031AH	Intel 8751H	4K	128	2	HMOS
Intel 8052AH	Intel 8032AH	Intel 8752BH	8K	256	3	HMOS
Intel 80C51BH	Intel 80C31BH	Intel 87C51	4K	128	2	CHMOS

The CHMOS devices draw less current than other devices in HMOS version. In addition, power saving features like **Idle Mode** and **Power Down Mode** are also introduced. Both the modes are controlled by the user software.

- In the idle mode, the CPU is turned off, whereas other devices like RAM and on-chip units remain active. Power drawn in the idle mode is 15% of full power.
- In the power down mode, all the on-chip activities are suspended. The on-chip RAM continues to hold the data. The device draws only 10 μ A current.

9.3 MEMORY ORGANIZATION

The 8051 can access up to 64 KB of program memory and 64 KB of external data memory and also the internal data RAM locations. Figure 9.2 shows the memory mapping in this chip.

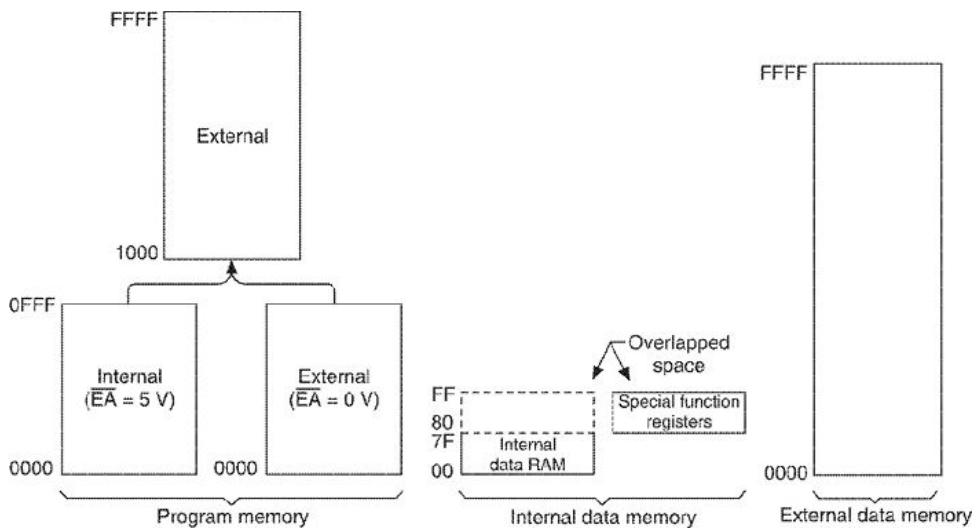


Figure 9.2 Memory mapping in the 8051.

The separation of code and data memory in the 8051 is different from the usual von Neumann architecture, which defines that code and data can share the common memory. The separated memory architecture is referred to as Harvard architecture.

9.3.1 Program Memory

It is clear from Figure 9.2 that 64 KB of program memory also includes the 4 KB of the on-chip ROM if used. Now, how will the processor know whether the 4 KB of on-chip ROM is being used or not? The answer is simple. The processor will come to know whether the user wants to use the internal ROM or not from the status of the \overline{EA} pin. If this pin is pulled low, it means that the user does not want to use the internal ROM available. Hence, the processor will access 0000–FFFFH from the external program memory. If this pin is held high, the processor will access addresses 0000–0FFFH from the internal ROM and as the address goes above 0FFFH, it will access the external program memory that is interfaced with it. In short,

If $\overline{EA} = 0 \text{ V}$

External program memory = 0000–FFFFH (64K)

If $\overline{EA} = 5 \text{ V}$

Internal program memory (ROM) = 0000–0FFFH (4K)

External program memory = 1000–FFFFH (60K)

9.3.2 Data Memory

The 8051 can access 64 KB of the external data memory and 128 bytes of the internal data RAM and also 21 special function registers.

For accessing the external data memory, the processor can either issue an 8-bit address or a 16-bit address. To access the internal data memory, the 8-bit address is used. This 8-bit address can provide address spaces for 256 locations. The lower 128 addresses (0 to 127) are used as 128 bytes on-chip RAM and the upper part of the address space, i.e. (128 to 255) is used to address the various special function registers. Internal RAM partitioning is explained in Figure 9.3.

As may be seen from Figure 9.3, the lowest 32 bytes (0–31) are reserved for 4 banks of 8 registers, each R0–R7, out of which one bank may be used at any time. Now the question that arises is, which bank of registers is being addressed at a particular time? The answer is that the working bank is specified in two bits of the Program Status Word Register (Figure 9.4).

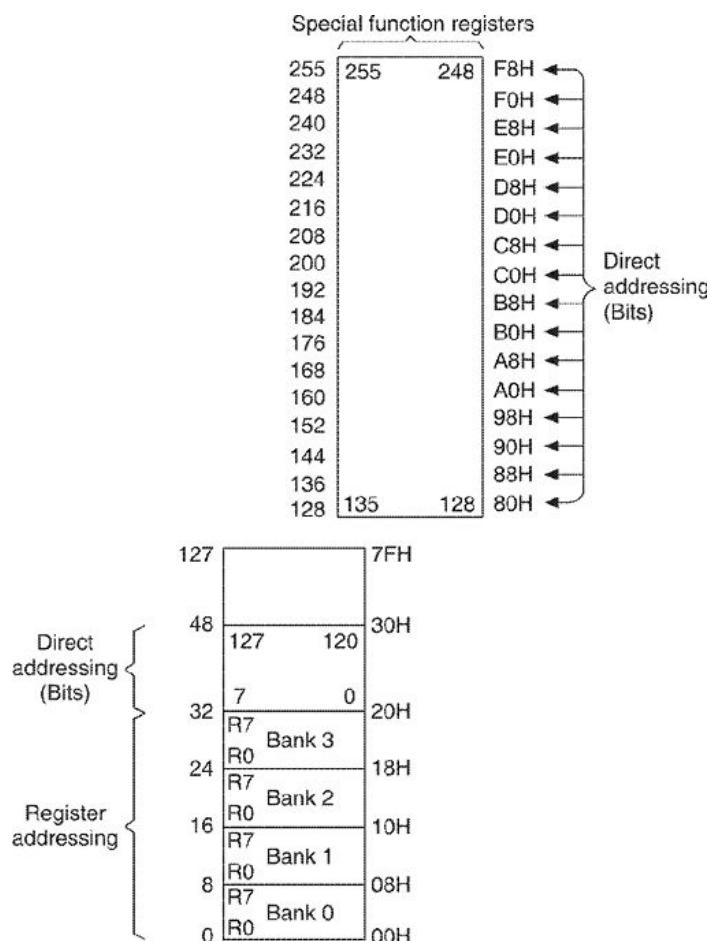


Figure 9.3 Internal RAM partitioning.

Bit no.	7	6	5	4	3	2	1	0
PSW	CY	AC	F0	RS1	RS0	OV	X	P

Where:

CY = Carry Flag
 AC = Auxiliary Carry Flag
 OV = Overflow Flag
 P = Odd Parity Flag
 F0 = User Flag 0

RS1	RS0	Register bank	
0	0	Register bank	0
0	1	Register bank	1
1	0	Register bank	2
1	1	Register bank	3

Figure 9.4 Program status word (PSW).

Memory locations 20H to 2FH are bit addressable too. The address of the bits in each byte is shown in Figure 9.5. Memory locations 30H–7FH are general purpose internal RAM locations. These bytes are not bit addressable.

RAM Byte (MSB)									(LSB)
7FH									127
2FH	7F	7E	7D	7C	7B	7A	79	78	47
2EH	77	76	75	74	73	72	71	70	46
2DH	6F	6E	6D	6C	6B	6A	69	68	45
2CH	67	66	65	64	63	62	61	60	44
2BH	5F	5E	5D	5C	5B	5A	59	58	43
2AH	57	56	55	54	53	52	51	50	42
29H	4F	4E	4D	4C	4B	4A	49	48	41
28H	47	46	45	44	43	42	41	40	40
27H	3F	3E	3D	3C	3B	3A	39	38	39
26H	37	36	35	34	33	32	31	30	38
25H	2F	2E	2D	2C	2B	2A	29	28	37
24H	27	26	25	24	23	22	21	20	36
23H	1F	1E	1D	1C	1B	1A	19	18	35
22H	17	16	15	14	13	12	11	10	34
21H	0F	0E	0D	0C	0B	0A	09	08	33
20H	07	06	05	04	03	02	01	00	32
1FH	Bank 3								31
18H									24
17H	Bank 2								23
10H									16
0FH	Bank 1								15
08H									8
07H	Bank 0								7
00H									0

Figure 9.5 Internal RAM bit address.

9.4 SPECIAL FUNCTION REGISTERS

All the resources in the 8051 can be accessed through special function registers maintained in the internal data memory. The resources like timers would require setting the modes and controlling them. Similarly, serial data input would require a serial data buffer as well as the control. The 8051 offers 4 ports which can be used for various purposes like input/output, memory interface, and so on. The functions of these resources can also be programmed through special function registers.

The mapping of special function registers in the internal data RAM is shown in Figure 9.6.

We shall now describe all the special function registers available in the 8051. As the name itself suggests, these registers have some special functions like controlling the timer/counter, enabling interrupts, controlling the serial port operations, etc. There are 21 special function registers in the 8051. Some special function registers are bit addressable. The various special function registers are shown below:

ACC	—	Accumulator*
B	—	B Register*
PSW	—	Program Status Word*
SP	—	Stack Pointer
DPTR (Low)	—	Data Pointer Low
DPTR (High)	—	Data Pointer High
P0	—	Port 0*
P1	—	Port 1*
P2	—	Port 2*
P3	—	Port 3*
IP	—	Interrupt Priority*
IE	—	Interrupt Enable*
TMOD	—	Timer/Counter Mode
TCON	—	Timer/Counter Control*
TH0	—	(Timer/Counter) 0 High
TL0	—	(Timer/Counter) 0 Low
TH1	—	(Timer/Counter) 1 High
TL1	—	(Timer/Counter) 1 Low
SCON	—	Serial Control*
SBUF	—	Serial Data Buffer
PCON	—	Power Control

*These bytes are bit addressable as well.

A brief description of each of the above mentioned special function registers is given below:

Accumulator: ACC (also called A register) is the accumulator. There are many instructions that use the accumulator as the destination. In nearly all arithmetic operations, ACC is used as one of the operands and after the operation, the result is also stored in the ACC.

Symbolic address	Bit address	Byte address
B	255 B 240	240 (F0H)
ACC	231 A 224	224 (E0H)
PSW	215 208	208 (D0H)
IP	191 184	184 (B8H)
P3	183 176	176 (B0H)
IE	175 168	168 (A8H)
P2	167 160	160 (A0H)
SBUF		153 (99H)
SCON	159 152	152 (98H)
P1	151 144	144 (90H)
TH1		141 (8DH)
TH0		140 (8CH)
TL1		139 (8BH)
TL0		138 (8AH)
TMOD		137 (89H)
TCON	143 136	136 (88H)
PCON		135 (87H)
DPH		131 (83H)
DPL		130 (82H)
SP		129 (81H)
P0	135 128	128 (80H)

Special function registers containing direct addressable bits

Figure 9.6 Mapping of the special function registers.

B Register: This register is mainly used for multiply and divide operations in which this register acts as one of the operands and after the operations a part of the result is stored in this register. This register otherwise is just like a scratch pad, i.e. a temporary register.

Stack Pointer: The stack pointer in the 8051 is an 8-bit wide register. This pointer can point to any location in the internal data RAM, i.e. locations from 0–127. When the chip is reset, this register is initialized to 07H. During PUSH and CALL instructions the stack pointer is first incremented and then the data is stored in the stack. That is, if initially (SP) = 20H, then the first bytes of the data will be stored at the 21H address.

EXAMPLE 9.1

Explain the stack operations.

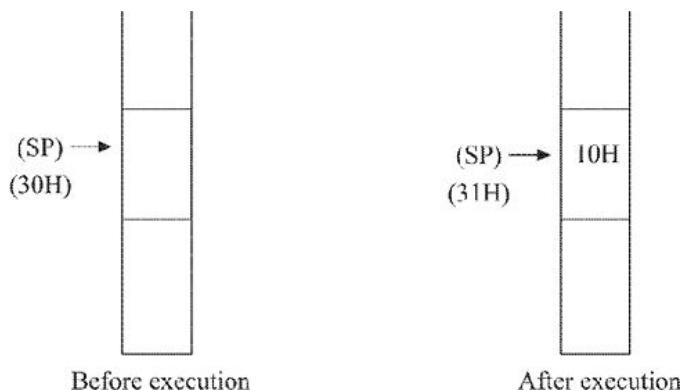
Solution:

(i) $(SP) = 30H$

Instruction—

DATAIN: DB 10H
 PUSH DATAIN

Operation— $(SP) \square (SP) + 1, ((SP)) \square (DATAIN)$
 $(SP) \square 31H, RAM(31H) \square 10H$

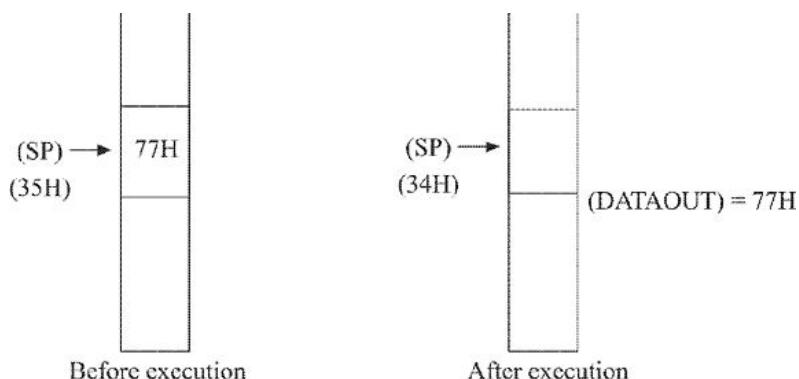


(ii) $(SP) = 35H$

Instruction—

DATAOUT: DB —
 POP DATAOUT

Operation— $(DATAOUT) \square ((SP)), (SP) \square (SP) - 1$
 $(DATAOUT) \square RAM(35H), (SP) \square 34H$
 $\square 77H$



Data Pointer: The data pointer (DPTR) is a 16-bit register, consisting of a high byte (DPH) and a low byte (DPL). This register normally contains a 16-bit address.

Ports 0–3: There are 4 bidirectional input/output ports of 8 bits each. These are directly represented as pins of the 8051 chips and are bit addressable. These are multifunctional resources that can be programmed based on the application needs, for example

- P0 and P2 can be used as I/O ports or address lines for external memory.
- P1 can be used as I/O port. It plays an important role in programming of internal memory of the 8751 and the 8051.
- P3 can be used as I/O port. Its pins have other important alternate functions like

Serial Input Line (P3.0), Serial Output Line (P3.1), External Interrupt Lines (P3.2, P3.3), External Timer Input Lines (P3.4, P3.5), External Data Memory Write Strobe (P3.6) and External Data Memory Read Strobe (P3.7)

These functions have been discussed in detail while describing the Pins of the 8051.

Timer 0 (TH0, TL0) and Timer 1 (TH1, TL1) are two 16-bit registers that can be used in timer/counter operations.

Serial Data Buffer (SBUF): This register holds the data that has to be transmitted through the serial port and also holds the data that is received. This register is interconnected to two 8-bit shift registers. When the data is written into the SBUF, it is loaded in the transmit shift register and hence the process of moving a data byte into the SBUF starts the transmission process. During reception, the data coming to the 8051 is clocked into the receive shift register and once all the 8 bits of data or a frame is received, it is transferred to the SBUF.

Control and status registers: All the special function registers (like IP, IE, TMOD, TCON, PCON, SCON, etc.) that are used for controlling the internal resources or to see the status of these resources come under this category. These registers contain the control and status bits of the interrupt systems, timers, and serial port operations. We will describe these registers in detail later in this chapter.

The bit addresses of various special function registers are shown in Figure 9.7.

The internal architecture of the 8051 is shown in Figure 9.8.

The instruction decoding and control signal generation are performed using the Timing and Control circuit block. The bidirectional ports P0 to

P3 have the basic structure containing drivers and latches. The port drivers are connected to I/O ports, whereas the latches are connected to the internal bus of the microcontroller.

The 8051 contains 128 bytes RAM as data memory and 4 KB ROM as program memory on the chip. Internal data RAM with address register is connected to the internal bus. The address register will get the address from the internal bus and the data will be transferred to the specified register through the bus.

Direct byte address	Bit addresses								Hardware register symbol
240	F7	F6	F5	F4	F3	F2	F1	F0	B
224	E7	E6	E5	E4	E3	E2	E1	E0	ACC
208	CY	AC	F0	RS1	RS0	OV		P	PSW
	D7	D6	D5	D4	D3	D2	D1	D0	
184				PS	PT1	PX1	PT0	PX0	IP
	—	—	—	BC	BB	BA	B9	B8	
176	B7	B6	B5	B4	B3	B2	B1	B0	P3
	EA		ES	ET1	EX1	ET0	EX0		
168	AF	—	—	AC	AB	AA	A9	A8	IE
160	A7	A6	A5	A4	A3	A2	A1	A0	P2
	SM0	SM1	SM2	REN	TB8	RB8	Ti	RI	
152	9F	9E	9D	9C	9B	9A	99	98	SCON
144	97	96	95	94	93	92	91	90	P1
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	
136	8F	8E	8D	8C	8B	8A	89	88	TCON
128	87	86	85	84	83	82	81	80	P0

Figure 9.7 Bit addresses of special function registers.

In order to access the internal program memory (ROM) as well as the external program memory, the program address register is interfaced to program counter which, in turn, is connected to the incrementer circuit for incrementing the PC. A bidirectional buffer has been provided to temporarily store the branch address as well as the operand address

information. This will also facilitate the transfer of the program counter contents to other circuits to calculate the PC relative jump address.

Ports P0 and P2 also facilitate the connection to the external memory. The lower-order address bits (A₀ to A₇) are sent through Port 0 (P0.0 to P0.7) and the higher order address bits (A₈ to A₁₅) are sent through Port 2 (P2.0 to P2.7). P0 also functions as data bus during external read/write in a time multiplexed manner which is same as in the 8085. Data pointer register (DPTR) is interfaced to the address register. It plays a major role in external program and data memory addressing using MOVC and MOVX instructions.

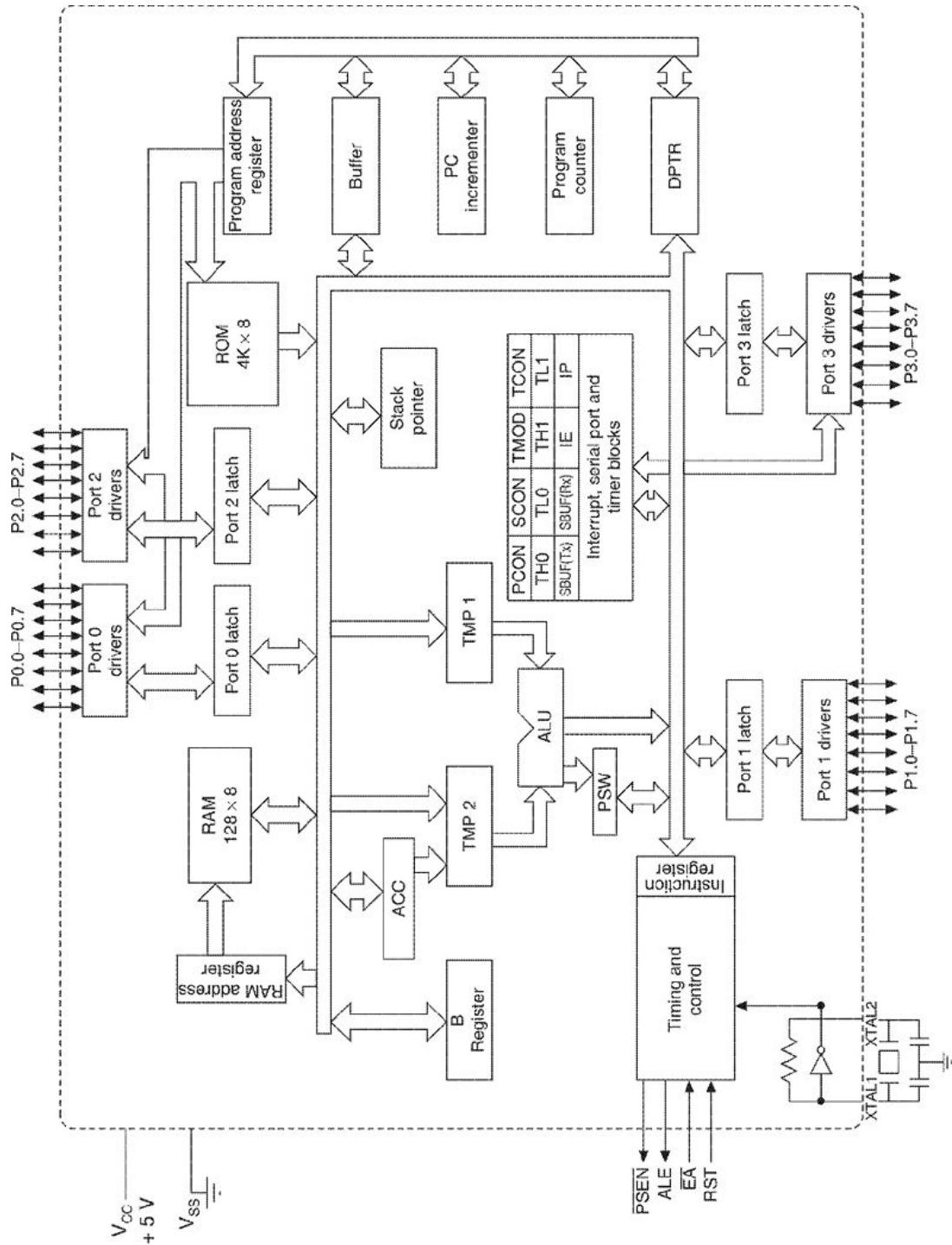


Figure 9.8 The 8051 architecture.

Two temporary registers TMP1 and TMP2 are connected to ALU. These registers hold the operands for calculation. The PSW register is directly interfaced to ALU for updation on the basis of the last operation.

Port 3 pins are used for input/output for serial I/O (P3.0 and P3.1), external interrupt inputs (P3.2 and P3.3), timer inputs (P3.4 and P3.5) and external data memory write and read strobes (P3.6 and P3.7). These are alternate function pins. Thus, Port 3 can be used purely as I/O port, if

these functions are not required. There is an alternate connection of Port 3 drivers to timer, interrupt and serial port circuit blocks. The specific special function registers to program and control these functions are contained in these circuit blocks.

9.5 PINS AND SIGNALS

Now we are ready to consider the 8051 at the pin level. A brief description of the pin out of the 8051 (Figure 9.9) is given below:

P1.0	1	40	V _{CC}
P1.1	2	39	P0.0 AD ₀
P1.2	3	38	P0.1 AD ₁
P1.3	4	37	P0.2 AD ₂
P1.4	5	36	P0.3 AD ₃
P1.5	6	35	P0.4 AD ₄
P1.6	7	34	P0.5 AD ₅
P1.7	8	33	P0.6 AD ₆
RST/VPD	9	8031	32 P0.7 AD ₇
RXD P3.0	10	8051	31 EA/VPP
TXD P3.1	11	8751	30 ALE/PROG
INT0 P3.2	12	29	PSEN
INT1 P3.3	13	28	P2.7 A ₁₅
T0 P3.4	14	27	P2.6 A ₁₄
T1 P3.5	15	26	P2.5 A ₁₃
WR P3.6	16	25	P2.4 A ₁₂
RD P3.7	17	24	P2.3 A ₁₁
XTAL2	18	23	P2.2 A ₁₀
XTAL1	19	22	P2.1 A ₉
V _{SS}	20	21	P2.0 A ₈

Figure 9.9 The 8051 pin diagram.

V_{CC} and V_{SS}: V_{CC} is the power supply pin. It is connected to a 5 V regulated supply. V_{SS} is the ground pin.

Port 0 (P0.0 to P0.7): These are 8-bit bidirectional input/output port pins. They are bit addressable. They can sink up to eight LS TTL logic gates. These pins also act as the lower part of the address bus as well as data bus while accessing the external memory. The lower part of the address bus is also time multiplexed with the data bus in the same way as in the 8085.

Port 1 (P1.0 to P1.7): Port 1 is an 8-bit bidirectional I/O port with internal pull-ups. That means an open collector output can be directly connected to this port without an external pull-up. It plays an important role during the programming of the internal memory of the 8051 and the 8751. This port can sink/source three LS TTL inputs.

Port 2 (P2.0 to P2.7): This is also an 8-bit bidirectional I/O with internal pull-ups. This also has an alternate function of higher-order address byte, while accessing the external memory. This also plays a role during the programming of the internal ROM of the 8051/8751. In all other features, it is same as Port 1.

Port 3 (P3.0 to P3.7): This port, like all others, is an 8-bit bidirectional I/O port with internal pull-ups. This also serves many other important alternate functions which enable the 8051 to be called a microcontroller. The alternate functions of Port 3 pins are listed below:

P3.0	—	RXD (Serial input)
P3.1	—	TXD (Serial output)
P3.2	—	$\overline{\text{INT0}}$ (External interrupt)
P3.3	—	$\overline{\text{INT1}}$ (External interrupt)
P3.4	—	T0 (Timer 0 external input)
P3.5	—	T1 (Timer 1 external input)
P3.6	—	$\overline{\text{WR}}$ (External data memory write strobe)
P3.7	—	$\overline{\text{RD}}$ (External data memory read strobe)

ALE/PROG: Address Latch Enable, also known as ALE, is an output pin and is used for latching the lower-order lines A₀–A₇ of the address bus as these lines are also time multiplexed with data lines. A pulse is generated at a constant rate of 1/6 oscillator frequency. This pulse is always generated except in a few cases like external data memory access, where one ALE pulse is skipped during each access. This pin also acts as an input pin for the programming pulses during the programming of the internal EPROM.

XTAL1 and XTAL2: The 8051 has an internal clock circuit, hence a crystal of proper frequency can be connected directly to these two pins. The frequency range is 3.5–12 MHz. An external oscillator can also be connected instead of a crystal. In this case, the XTAL1 pin is grounded and the oscillator signal is given to the XTAL2 pin.

PSEN: This pin gives out active low output pulses. This signal is used for fetching the data from the external program memory. A pulse is generated after every six clock cycles. This is used as a read signal for reading from the external program memory. If the data to be fetched is inside the chip itself, then PSEN is not generated. This signal is also not generated during the external data memory operation.

EA/VPP: This pin, if connected to 5 V, will indicate to the processor that the 4 KB of the program memory within the chip should be used. Hence,

addresses 0000H to 0FFFH will be within the chip. Once the address exceeds this, i.e. 1000H to FFFFH, the external program memory is accessed. This pin, if pulled low, will inform the processor that the on-chip ROM is not being used. Hence the entire 64 KB will be external to the chip. So, depending on whether a designer is using the internal program memory or not, he/she will have to connect the \overline{EA} pin to either 5 V or ground. This pin is also used while programming the 8751 as programming supply (VPP) voltage (21 V).

RST/VPD: A high on this pin, for more than 24 oscillator cycles, will reset the chip. VPD may be used to supply power to the internal RAM during power failure or power down modes.

9.6 TIMING AND CONTROL

The clock generating circuit in the 8051 divides the oscillator clock by 2 and provides a two-phase clock to the CPU (Figure 9.10). Phase-1(P1) will be active during the first-half of each of the clock periods and phase-2 (P2) will be active during the second-half.

A machine cycle consists of six internal clock periods S1 to S6 (also called states) or 12 oscillator clock cycles. Normally, all arithmetic and logic operations take place during Phase-1. Internal register-to-register transfers take place during Phase-2.

Let us now discuss some of the signals that the CPU regularly generates.

ALE is generated twice in each machine cycle, once during S1P2 and S2P1, and the second time during S4P2 and S5P1. This signal is used for latching the lower order address since the lower order address bus is time multiplexed with the data bus. However during machine cycle of external data memory access, ALE is not generated.

PSEN is also sent by the CPU to the external program memory (in general) to read a byte of a code. **PSEN** becomes active twice in a machine cycle, once during S3 and the second time during S6. So if an instruction is two bytes long, then the first byte is fetched by the first and the second byte is fetched by the second **PSEN** pulse.

Now, what will happen if the instruction is only one byte long? The answer is simple. The CPU will discard the second code read during the second **PSEN** pulse. When the code byte read is not used (discarded), the program counter is not incremented and in the next cycle CPU will again read the same memory location.

Another question that arises is what will happen to the PSEN when the code is being fetched from the internal program memory? The PSEN signal is not activated during all internal fetches

What will happen if the instruction requires external data memory access?

In this case, the instruction is read and decoded in the first machine cycle. Then RD, instead of PSEN, is activated. The code read during S4 of the first machine cycle is discarded. In the second machine cycle of this instruction, RD is made active and the ALE during S2 is missed, and by the end of the second machine cycle, the data will be fetched from the external data memory. The only example of such an instruction is **MOVX** (Figure 9.11).

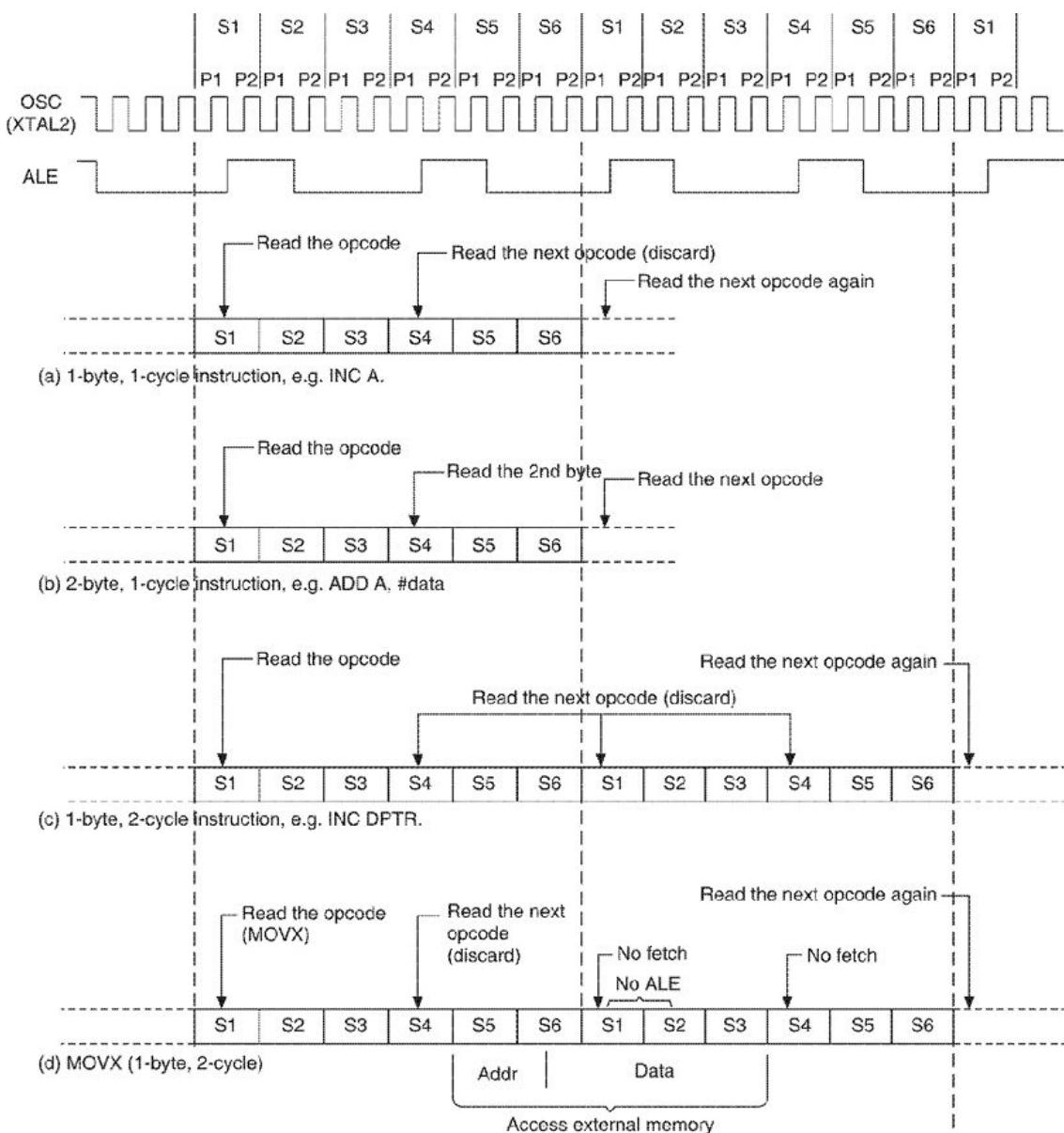


Figure 9.10 Fetch/execute sequences—timing diagram.

Figure 9.10 shows the fetch and execute sequence for four different types of instructions like 1-byte, 1-cycle instruction, e.g. INC A (Figure 9.10(a)); 2-byte, 1-cycle instruction, e.g. ADD A, # data; (Figure 9.10(b)); 1-byte, 2-cycle instruction, e.g. INC DPTR (Figure 9.10(c)) and 1-byte, 2-cycle instruction with external data memory access, e.g. MOVX (Figure 9.10(d)). $\overline{\text{PSEN}}$, the read signal for external program memory is not shown. We may assume that Figure 9.10 shows instruction execution from internal program memory.

The execution of instructions from external program memory is shown in Figure 9.11. Whereas Figure 9.11(b) shows the timing diagram for MOVX instruction, Figure 9.11(a) shows it for all other instructions. It must be understood that the operation sequence and time taken for any instruction to execute is same irrespective of whether it is in internal or external program memory.

EXAMPLE 9.2

Let us assume that the external program memory locations XXY0 to XXY4 store the following:

XXY0	-	Code for INC A
XXY1	-	Code for ADD A, #Data
XXY2	-	Data
XXY3	-	Code for INC DPTR
XXY5	-	Code for MOVX A, @ DPTR

Work out the execution sequence of these instructions. These are the same instructions whose fetch/execute sequence is shown in Figure 9.10.

Solution:

Let us say that the above instructions are executed from Machine Cycle MC1 onwards. We shall now work out the execution sequence (the instruction decoding and control signal generation, and exact execution are not described).

State	Signal	Remarks
MC1-1 (machine cycle prior to MC1)		
S5	(P2) = XX, (P0) = Y0 $\overline{\text{PSEN}}$ = high, ALE = high to low	XXY0 = Address of the next instruction P0 contents are latched.
S6	$\overline{\text{PSEN}}$ = low	External program memory read initiated
MC1 (machine cycle 1)		
S1	(P0) = Instruction Code for INC A (P2) = XX, $\overline{\text{PSEN}}$ = high, ALE = high	Instruction stored in Instruction Register
S2	(P2) = XX, (P0) = Y1	XXY1 = Address of the next instruction P0

	$\overline{\text{PSEN}} = \text{high}$, ALE = high to low	contents are latched
S3	ALE = low, $\overline{\text{PSEN}} = \text{low}$	External program memory read initiated
S4	(P0) = Instruction code of ADD A, # Data (P2) = XX, ALE = high $\overline{\text{PSEN}} = \text{high}$	Since program memory read is not required for execution of the instruction in progress, it is discarded
S5	(P2) = XX, (P0) = Y1 $\overline{\text{PSEN}} = \text{high}$, ALE = high to low	XXY1 = Address of the next instruction P0 contents are latched
S6	$\overline{\text{PSEN}} = \text{low}$	External program memory read initiated
MC2 (machine cycle 2)		
S1	(P0) = Instruction code for Add A, # Data (P2) = XX, $\overline{\text{PSEN}} = \text{high}$ ALE = high	Instruction is sent to Instruction Register and decoded. Since immediate data is required to be read, it waits for the next program memory read.
S2	(P2) = XX, (P0) = Y2 $\overline{\text{PSEN}} = \text{high}$, ALE = high to low	XXY2 = Address of immediate data. P0 contents are latched
S3	ALE = low, $\overline{\text{PSEN}} = \text{low}$	External program memory read initiated.
S4	(P0) = Databyte, (P2) = XX ALE = high, $\overline{\text{PSEN}} = \text{high}$	Execution of instruction takes place.
S5	(P2) = XX, (P0) = Y3 $\overline{\text{PSEN}} = \text{high}$, ALE = high to low	XXY3 = Address of the next instruction P0 contents are latched
S6	$\overline{\text{PSEN}} = \text{low}$	External program memory read initiated
MC3 (machine cycle 3)		
S1	(P0) = Instruction code for INC DPTR (P2) = XX, $\overline{\text{PSEN}} = \text{high}$, ALE = high	Instruction sent to Instruction Register and decoded
S2	(P2) = XX, (P0) = Y4 $\overline{\text{PSEN}} = \text{high}$, ALE = high to low	XXY4 = Address of the next instruction P0 contents are latched
S3	ALE = low, $\overline{\text{PSEN}} = \text{low}$	External memory read initiated
S4	(P0) = Instruction code of MOVX A, @ DPTR (P2) = XX, ALE = high, $\overline{\text{PSEN}} = \text{high}$	Since execution of the previous instruction is in progress and no program memory read is required, it is discarded.
S5	(P2) = XX, (P0) = Y4 $\overline{\text{PSEN}} = \text{high}$, ALE = high to low	XXY4 = Address of the next instruction P0 contents are latched
S6	$\overline{\text{PSEN}} = \text{low}$, ALE = low	External program memory read initiated
MC4 (machine cycle 4)		
S1	(P0) = Instruction code of MOVX A, @ DPTR (P2) = XX ALE = high, $\overline{\text{PSEN}} = \text{high}$	Instruction is discarded as the previous instruction execution is in progress
S2	(P2) = XX, (P0) = Y4 $\overline{\text{PSEN}} = \text{high}$, ALE = high to low	XXY4 = Address of the next instruction P0 contents are latched
S3	ALE = low, $\overline{\text{PSEN}} = \text{low}$	External program memory read initiated
S4	(P0) = Instruction code of MOVX A, @ DPTR (P2) = XX, ALE = high, $\overline{\text{PSEN}} = \text{high}$	Instruction is discarded as the previous instruction execution is in progress

S5	(P2) = XX, (P0) = Y4 $\overline{\text{PSEN}}$ = high, ALE = high to low	XXY4 = Address of the next instruction P0 contents are latched
S6	$\overline{\text{PSEN}}$ = low	External program memory read initiated
MC5 (machine cycle 5)		
S1	(P0) = Instruction code of MOVX A, @ DPTR (P2) = XX, ALE = high, $\overline{\text{PSEN}}$ = high	Instruction set to Instruction Register and decoded
S2	(P2) = XX, (P0) = Y5 $\overline{\text{PSEN}}$ = high, ALE = high to low	XXY5 = Address of the next instruction P0 contents are latched
S3	ALE = low, $\overline{\text{PSEN}}$ = low	External program memory read initiated
S4	(P0) = Instruction code of next instruction (P2) = XX, ALE = high $\overline{\text{PSEN}}$ = high	Discarded since instruction execution does not require read from program memory
S5	(P2) = (DPH), (P0) = (DPL) $\overline{\text{PSEN}}$ = high $\overline{\text{RD}}$ = high ALE = high to low	P0 contents are latched
MC6 (machine cycle 6)		
S1	$\overline{\text{PSEN}}$ = not generated $\overline{\text{RD}}$ = low	External data memory read initiated
S2	(P0) = Data, (P2) = (DPH) $\overline{\text{RD}}$ = low, ALE = not generated	External data read
S3	$\overline{\text{RD}}$ = high	Data placed in register
S4	ALE = high, $\overline{\text{PSEN}}$ = high, $\overline{\text{RD}}$ = high	
S5	(P2) = XX, (P0) = Y5 $\overline{\text{PSEN}}$ = high, ALE = high to low	XXY5 = Address of the next instruction P0 contents are latched
S6	$\overline{\text{PSEN}}$ = low	External program memory read initiated
MC7 (machine cycle 7)		
S1	(P0) = Instruction code of next instruction (P2) = XX, ALE = high, $\overline{\text{PSEN}}$ = high — — —	Instruction is put in instruction register and decoded

You may like to compare the above with the fetch execute sequence in Figure 9.10 and the instruction execution sequence in Figure 9.11.

Most instructions in the 8051 take one machine cycle. There are only two instructions that require more than two machine cycle and they are MUL and DIV. These two instructions take four machine cycles.

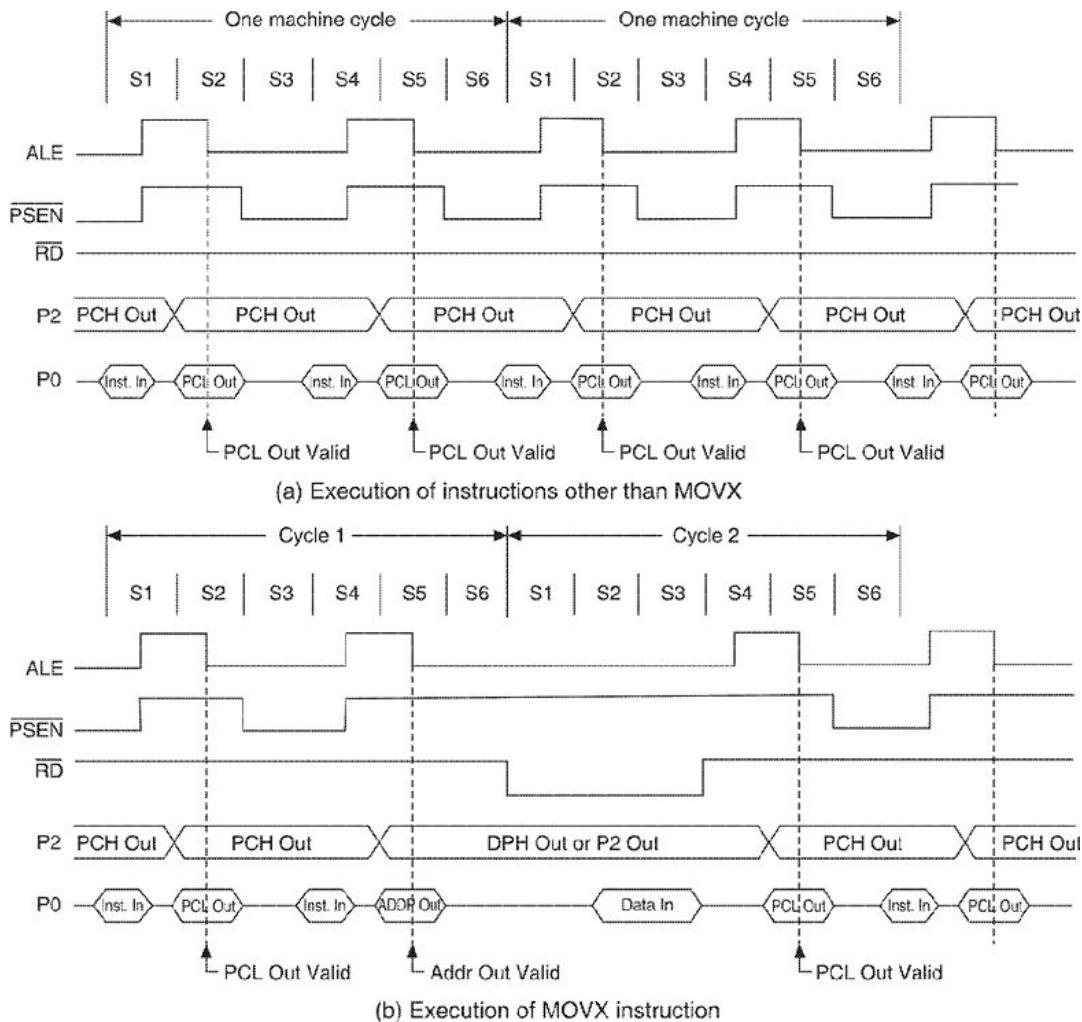


Figure 9.11 Instruction execution from external program memory.

Resetting the processor

Reset is accomplished by holding the RST pin high for at least two machine cycles (24 oscillator periods) while the oscillator is still running. The CPU responds by executing an internal reset. It also configures the ALE and PSEN pins as inputs. (They are quasi-bidirectional.) The internal reset is executed during the second cycle in which RST is high and is repeated in every cycle until RST goes low. It leaves the internal registers as follows.

Register	Content
PC	0000H
A	00H
B	00H
PSW	00H
SP	07H
DPTR	0000H
P0–P3	0FFH
IP	(XXX00000)

IE	(0XX00000)
TMOD	00H
TCON	00H
TH0	00H
TL0	00H
TH1	00H
TL1	00H
SCON	00H
SBUF	Indeterminate
PCON	(0XXXXXXX)

9.7 PORT OPERATION

As mentioned earlier, the 8051 has four bidirectional ports P0 to P3. Figures 9.12 to 9.15 show the functional diagrams of a typical bit in these ports. Whereas P1, P2, P3 have internal pull-ups, P0 has open-collector outputs. Each I/O line can be independently used as either input or output line. Thus it is possible to use any port line as input or output line.

As shown in Figure 9.8 and also Figures 9.12 to 9.15, each port line contains a latch, a driver and an input buffer. Whenever an input operation is required, a 1 is stored in the latch corresponding to the port line.

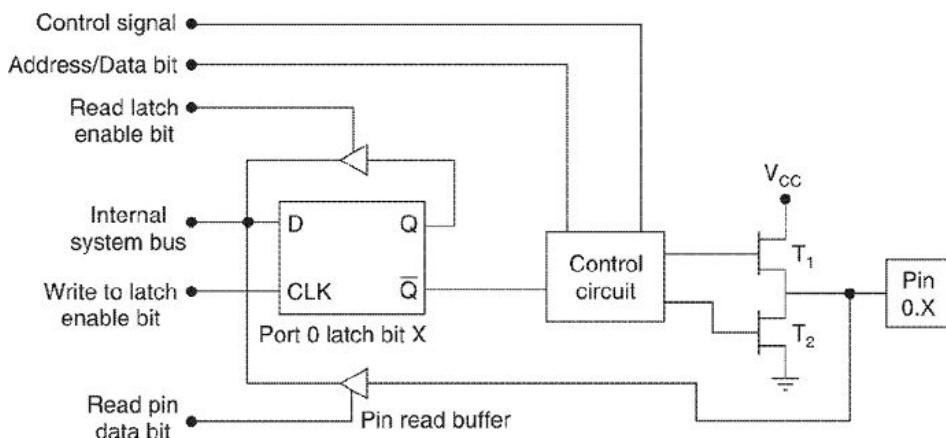


Figure 9.12 Port 0 configuration.

Thus to configure a port Pk for input

MOV A, #FFH

MOV Pk, A (where k = 0 to 3), may be used

Examples of the instructions for the input operations are MOV A, P2; MOV C, P3.2, etc. To configure port k line no ‘n’ for input—SETB Pk □ n (where k = 0 to 3, n = 0 to 7) may be used. Similarly, instructions like MOV P3, #50H; MOV P3.3, C will correspond to the output

operation. Each of the output buffers of Port 0 can drive up to eight LSTTL inputs, whereas output buffers of Ports 1, 2 and 3 can drive up to three LSTTL inputs.

9.7.1 Port 0

Port 0 is (Figure 9.12) an 8-bit open collector bidirectional I/O port. It can perform two functions:

- Simple I/O operation
- External memory interface for lower-order address bus and data bus.

Simple I/O operation

When a 1 is loaded to the latch as part of the input operation, both the FETs, T₁ and T₂, are turned off. This causes Port 0 pin to float to high impedance state and it gets connected to the Pin Read Buffer. An external pull-up register is required to supply a high output. Thus, the port is configured for input operations. Information on the P0 pin is transferred to internal bus when the Read Pin signal is generated.

When used as an output port, data will be directly loaded to latch. If a 0 is the output on the latch, it will turn on FET T₂ and thus the output pin will be grounded. If a 1 is being output, the output pin will float to high impedance state and the external pull-up register may be used to output 1 as the data state.

External memory interface

For any read/write to the external memory, first the address and then the data transfer will be done. When the address is to be transferred to the pin, the control signals will connect the address information directly to pin through FETs T₁ and T₂. The port latch is disconnected in this case.

If the address bit = 1, then T₁ will be turned on and T₂ will be turned off. This will cause logic 1 signal on the port pin. When the address bit = 0, then T₁ is turned off and T₂ is turned on. This will ground the output pin and provide the logic 0 signal.

After the transfer of the address information, if memory read is required then the 8051 places a 1 on the latch and thus reconfigures the pin to receive data.

9.7.2 Port 1

Port 1 is a bidirectional (Figure 9.13) I/O port with internal pull-ups. The circuit has three FETs T₁, T₂ and T₃. FETs T₂ and T₃ work as internal pull-up to T₁.

When Port 1 is used as input, logic level 1 is loaded to the latch. This will turn off the FET T₁. This will effectively float the port pin to high impedance state and will be connected to the Pin Read Buffer. The pin is at logic level 1 at this instant. An external device may place a 0 by driving the pin to the ground, or it may place a 1 on the pin by leaving it at logic state 1.

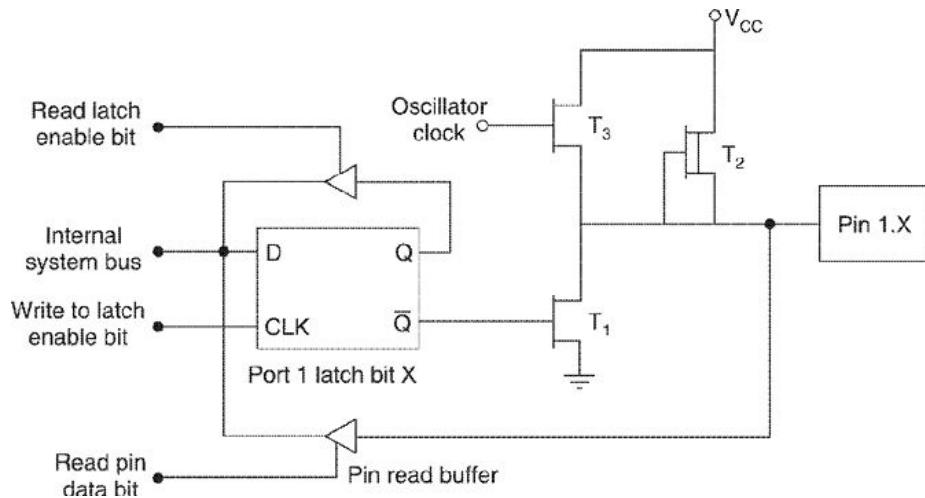


Figure 9.13 Port 1 configuration.

When used as an output port, the value will be loaded to the latch. The latches which contain level 1 will turn off FET T₁ and this will drive the pin high through the pull-up register formed by T₂ and T₃. The latches containing level 0 will turn on the FET T₁. This will ground the port pin and it will show logic level 0.

The FETs T₂ and T₃, forming internal pull-up, are depletion type and enhancement type respectively. The FET T₃ is turned on for two clock pulses which provide low impedance path to the supply voltage. This reduces transients in the external circuit and helps in high-speed operation.

9.7.3 Port 2

Port 2 (Figure 9.14) can be used as a bidirectional I/O port or it can also be used to output higher-order address bus for external memory interface.

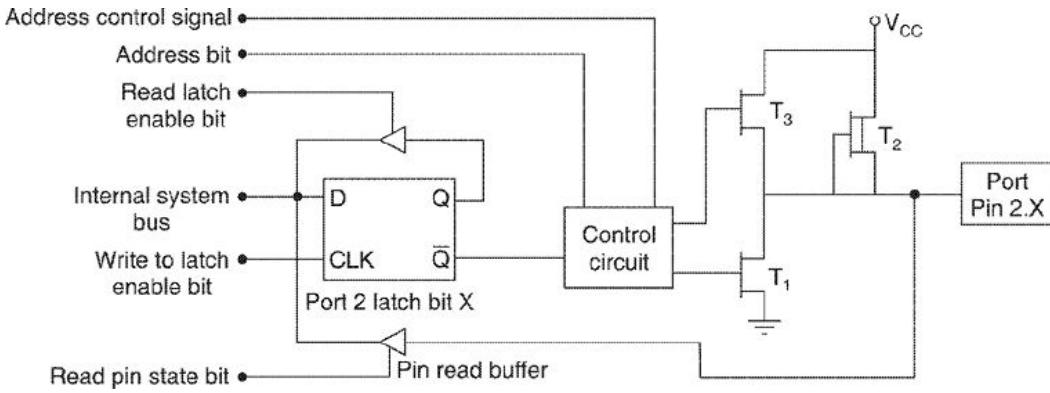


Figure 9.14 Port 2 configuration.

When used as an input port, logic level 1 is stored in the latch. The FET T₁ is turned off and floats in the high impedance state. The pin is directly connected to Pin Read Buffer. The external device can place a 0 or a 1 on the pin.

When used as an output port, the working of Port 2 is similar to Port 1. That means that the latches containing 0 will turn on FET T₁, thus grounding the pin. The latches containing 1 will turn off T₁ and thus the pin will be driven to logic level 1.

When Port 2 is used to output higher-order address bits for external memory access, the control signal will bypass the latch and the address bits are directly connected to the pin through FETs.

9.7.4 Port 3

Port 3 (Figure 9.15) is a bidirectional I/O port with internal pull-up. Different pins of Port 3 also facilitate alternate functions. The function of Port 3 pins is either under the control of the port latches or under the control of the special function registers. These pins are individually programmable unlike other ports where alternate functions of all port pins are programmed together.

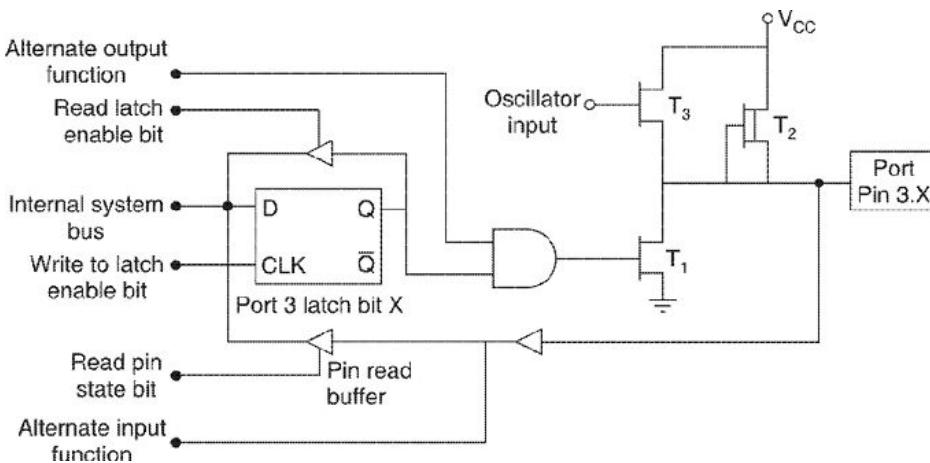


Figure 9.15 Port 3 configuration.

Following are the alternate functions of different pins of Port 3.

P3.0	Serial data input (RXD)
P3.1	Serial data output (TXD)
P3.2	External interrupt 0 ()
P3.3	External interrupt 1 ()
P3.4	Timer 0 external input (T0)
P3.5	Timer 1 external input \overline{WR} (T1)
P3.6	External data memory write strobe \overline{RD} ()
P3.7	External data memory read strobe ()

It is evident from the structure of the ports depicted in Figures 9.12 to 9.15, that a port may be read in two ways:

- read latch using Read Latch signal
- read pin using Read Pin signal

Some instructions in the 8051 read the latch, whereas some other instructions read the pin. The instructions which read the latch are those that read a value and write it back with some change or without any change.

Such instructions are described as Read-Modify-Write instructions. Read-Modify-Write instructions are directed to the latch since the voltage level at the pin may sometimes be wrongly interpreted.

As an example, suppose a port pin is connected to the base of a transistor. The transistor may be turned on by writing a 1 to the pin. When the transistor is turned on, the base voltage will come down and if an instruction reads the pin, it will read a zero value. At the same time, if the instruction reads the latch, a value 1 will be read.

The instructions of the 8051 which come under the category are:

MOV	PX.Y, C (Move carry to bit Y of Port X)
INC	PX (Increment the contents of Port X and store back)
DEC	PX (Decrement the contents of Port X and store back)
ANL	PX, <SRC> (Logically AND the Port X contents with the source operand and store back the result to the port)
ORL	PX, <SRC> (Logically OR the Port X contents with the source operand and store back the result to the port)

XRL PX, <SRC> (Logically EX-OR the Port X contents with the source operand

 and store back the result to the port)

CPL PX.Y (Complement bit Y of Port X and store back)

CLR PX.Y (Clear bit Y of Port X and store back)

SET PX.Y (Set bit Y of Port X and store back)

JBC PX.Y, LABEL (Jump to LABEL if bit Y of Port X = 1
and clear the bit)

DJNZ PX, LABEL (Decrement Port X contents and jump to
LABEL if not zero)

This aspect has been mentioned while describing the instructions in the next chapter.

In the execution of the instructions, which change the value in the port latch, the new value arrives at latch during S6 P2 of the final cycle of the instruction. However, port buffers sample the latches only during phase P1 of any clock-period. Thus, the new value in the latch will appear only during S1P1 of the next instruction machine cycle.

9.8 MEMORY INTERFACING

To interface a memory chip to the microprocessor, the following signals are required to be generated.

- Address Bus Signals
- Data Bus Signals
- Memory chip Select Signals
- Read Control Signal
- Write Control Signal (only in case of RAM).

Let us now see, how these signals are mapped in the 8051 microcontroller environment.

- Address and data bus signals are provided through alternate functions in Ports P0 and P2 pins.
 - Pins P2.0 to P2.7 constitute A₈ to A₁₅, i.e. higher-order address lines.
 - Pins P0.0 to P0.7 represent both data lines (D₀–D₇) and lower-order address lines A₀ to A₇, in the time-multiplex way.
 - As in case of the 8085, an external latch will be required to

latch the lower address byte when present. The latch is strobed through ALE (Address Latch Enable) signal. The 8282 latch chip (described in Chapter 5) is generally used.

- The memory chip select signals are obtained by decoding the address information, through a decoder. The chip select signal to a memory chip signifies that the address in question is contained in the chip and thus the read/write operation will be performed on the chip selected.

The address space required for the application is decided and then, the same is divided into Program and Data memory as well as RAM and ROM. The address space for each chip is decided, and based on this, the decoder is connected to different address lines. The 74LS138 decoder chip is popularly used for address decoding. We shall attempt to interface the following memory chips to the 8051.

<i>Memory chip</i>	<i>Address range</i>
2732/6132(1)	0000H to 0FFFH(4K)
2716/6116(2)	1000H to 17FFH (2K)
2716/6116(3)	1800H to 1FFFH(2K)
2716/6116(4)	2000H to 27FFH(2K)

- For External Data Memory, \overline{RD} (P3.7) and \overline{WR} (P3.6) are used as read and write control signals. Figure 9.16 shows the interfacing of the external data memory to the 8051. Access to the external data memory can use either a 16-bit address (e.g. MOVX A, @DPTR) or an 8-bit address (e.g. MOVX A, @Ri). If the 8-bit address is being used, then the Port 2 contents remain unchanged and the Port 0 contents provide the address information. This facilitates the paging of the external data memory.

Only ROM is allowed as the external program memory. \overline{PSEN} (Program Store Enable) acts as the read control signal to access the memory. Figure 9.17 shows the interfacing of the external program memory to the 8051.

Many applications require the 8051 to be used as a general-purpose microprocessor. Such applications demand that the external program memory may be implemented through RAM. In addition, the system development phase often requires continuous program modification, thus requiring RAM as the program memory. This can be achieved by merging the external

data memory and the external program memory spaces into one single 64 KB memory space, used for data as well as program. Figure 9.18 shows the interfacing of the combined external program and data memories to Intel 8051. \overline{WR} is used as the write control signal. \overline{PSEN} and \overline{RD} are ANDed to provide the read control signal.

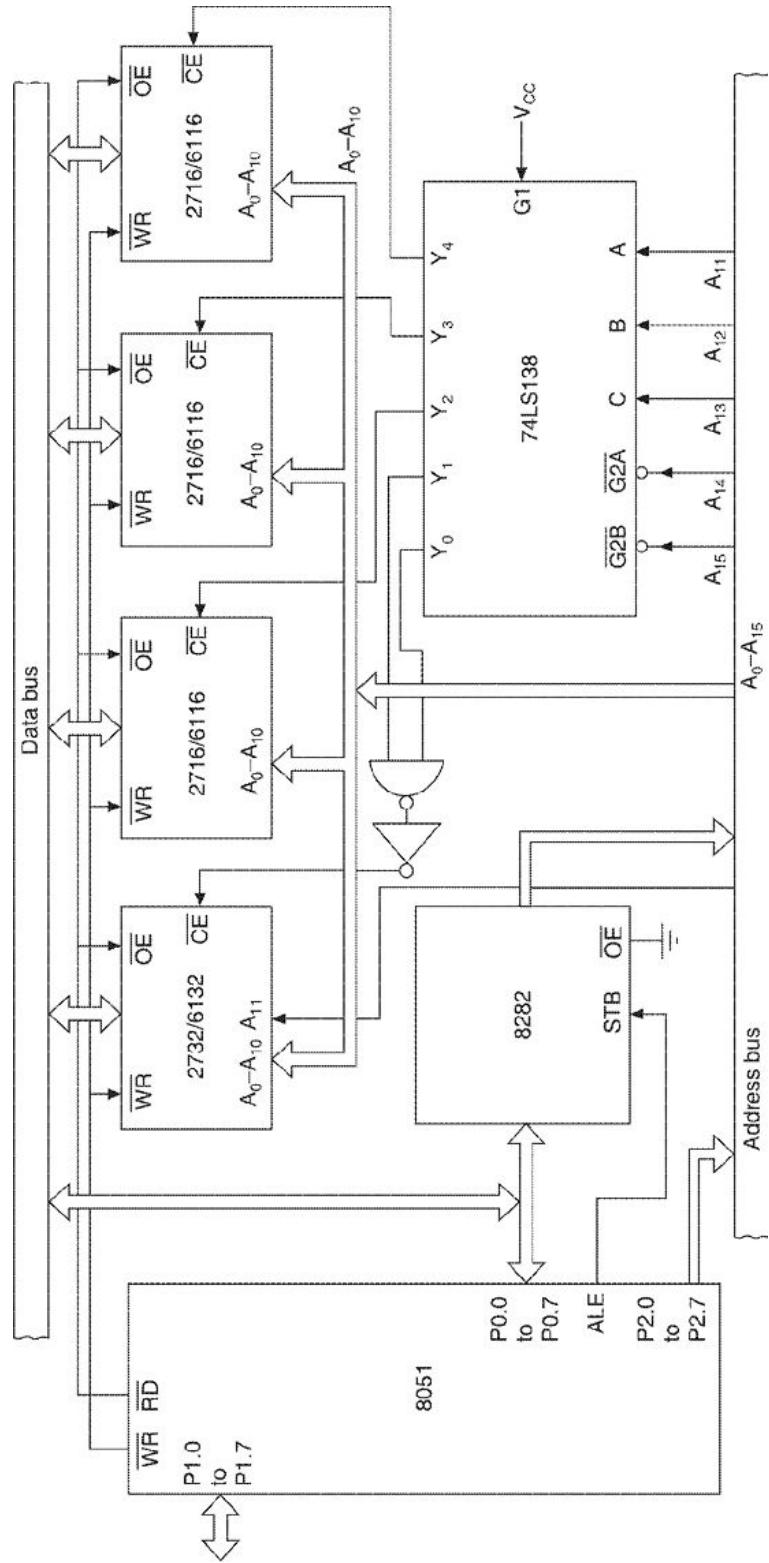


Figure 9.16 External data memory interface.

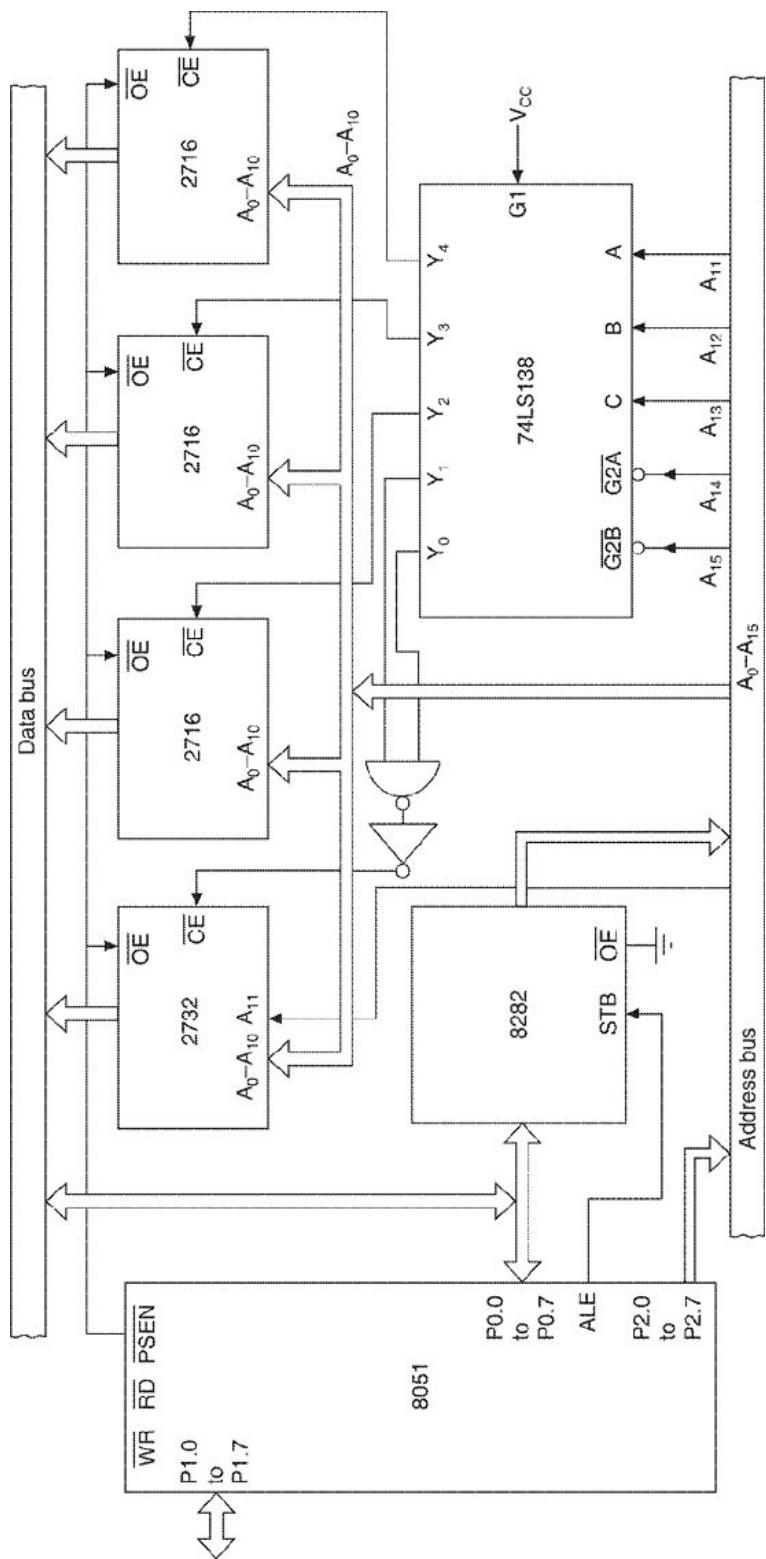


Figure 9.17 External program memory interface.

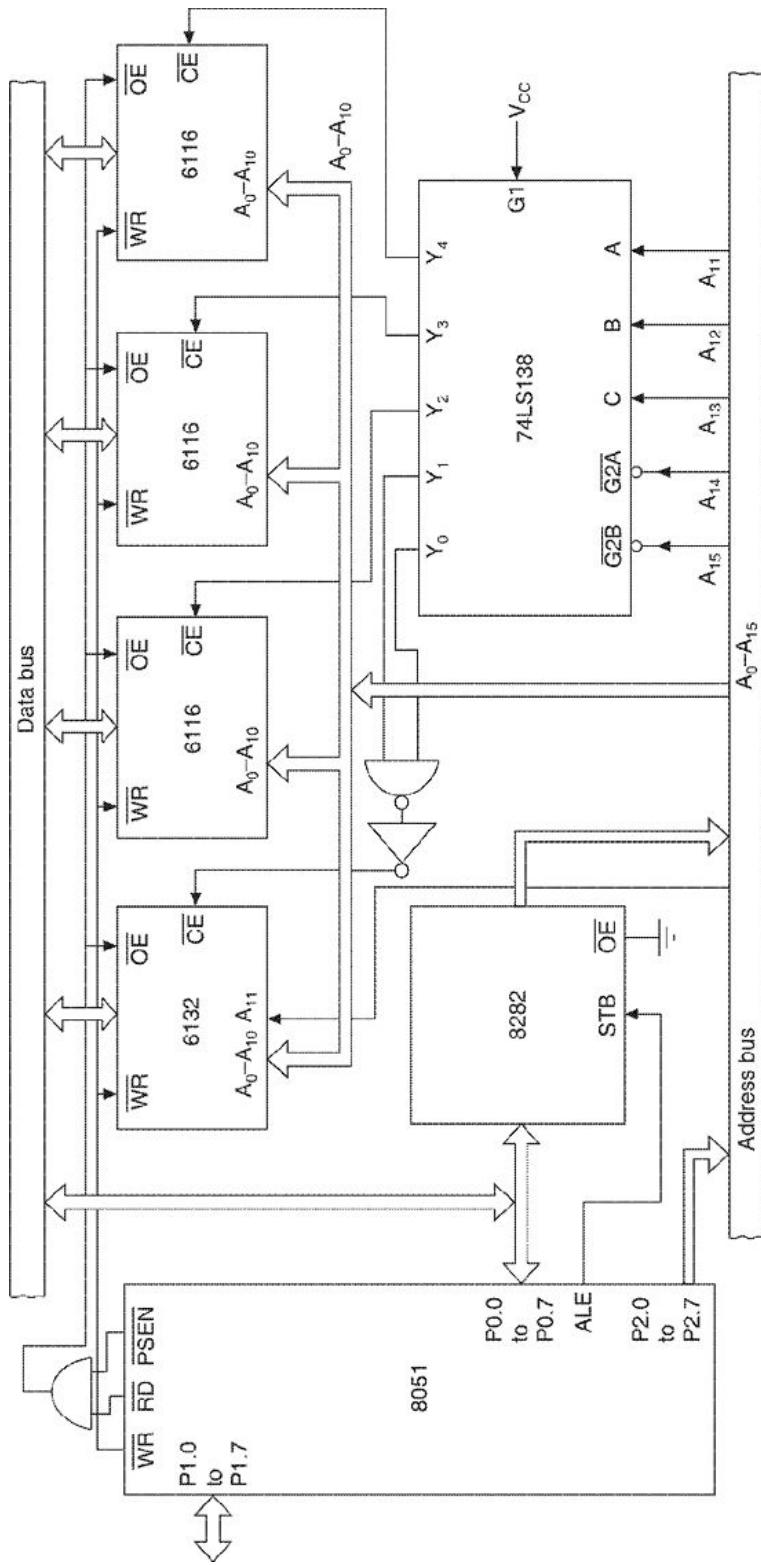


Figure 9.18 Combined external program and data memory interface.

9.9 I/O INTERFACING

The 8051 architecture provides four I/O ports which can be effectively used for interfacing of any input-output device. It must, however, be

understood that these ports have certain alternate functions as well, which may need to be sacrificed.

Figure 9.19 shows a simple interface of a 4 × 4 keyboard, with the 8051 using one of the ports—let us say P1.

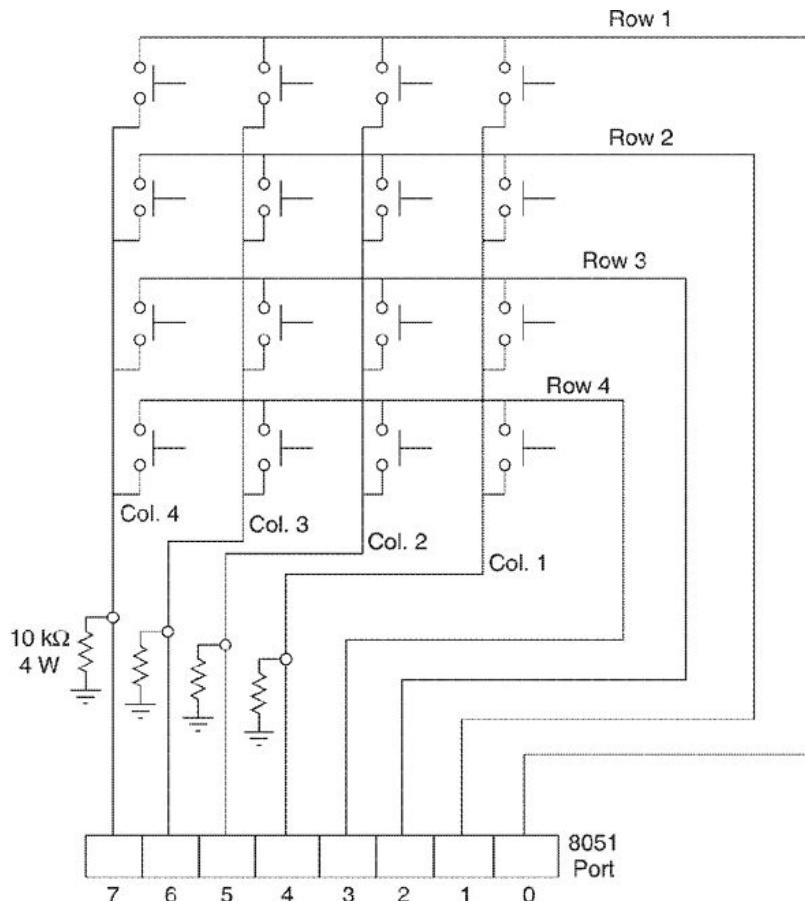


Figure 9.19 4 × 4 keyboard interface.

The keyboard (refer Section 7.5.1 for details) has been divided into rows and columns, and connected to the port lines in the following manner:

Row 1 to Row 4	Port lines 0 to 3
Column 1 to Column 4	Port lines 4 to 7

A particular key, when pressed, can be identified by its row and column numbers. However, to find out whether a particular key has been pressed or not, the 8051 will have to scan the keys regularly. Algorithm for this is simple.

```

For row no.  $j = 1$  to 4
  Send level 1(5 V) signal to row no.  $j$ 
  For column no.  $k = 1$  to 4
    Read signal level at column no.  $k$ 
    If level = 1, then go to keycode
  
```

```
else repeat column  
repeat row
```

Keycode: Input code for key (j, k).

Like keyboard, the LED display will often be used in applications requiring the 8051 as stand-alone system. These applications may differ from a simple traffic light controller to a complex process controller. Single LEDs can be used for ON/OFF indication or alarm annunciation.

Figure 9.20 shows the interfacing of eight LEDs (refer Section 7.5.2 for details) to the 8051 through a port in the common anode fashion. A level 0 at any port line will glow the LED connected to the line. Alternate 0 and 1 on the line, with some wait period in between, will enable the blanking of LED and this may be used for alarm indication.

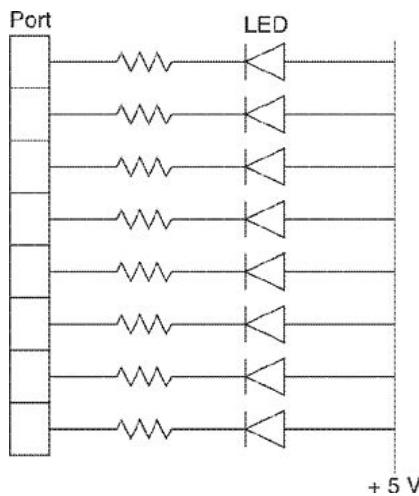
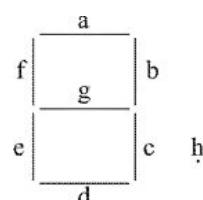


Figure 9.20 LED interface (common anode).

The seven-segment LED displays are very popular in instruments, industry as well as various appliances, and are used to display decimal digits. The seven-segment display looks as shown below, where a, b, c, d, e, f, g, h are called the segments of the seven-segment display.



In fact, it is eight-segment display as the eighth segment 'h' has been added to display the decimal point. The coding of the digit to be displayed is done as follows:

Bit no.	7	6	5	4	3	2	1	0
Segment	h	g	f	e	d	c	b	a

Whichever segment we want to glow, we store binary ‘0’ at the particular position of the segment. (This is applicable in case of the common anode type seven-segment display, whereas in the common cathode, it is just the reverse.)

Now, suppose we want to display 2 as per the standard segment nomenclature, then, the segments to glow will be a, b, g, e and d. If we are using the common anode type, we will store ‘0’ at those locations. So the equivalent code would be 25H. The display format for other digits can be worked out in the same manner.

The serial interface of seven-segment LED display is shown in Figure 9.21. The interface circuit contains four modules of the seven-segment display, connected in serial mode through the 74164 shift register. The 8-bit display code for the modules can be stored in four memory locations.

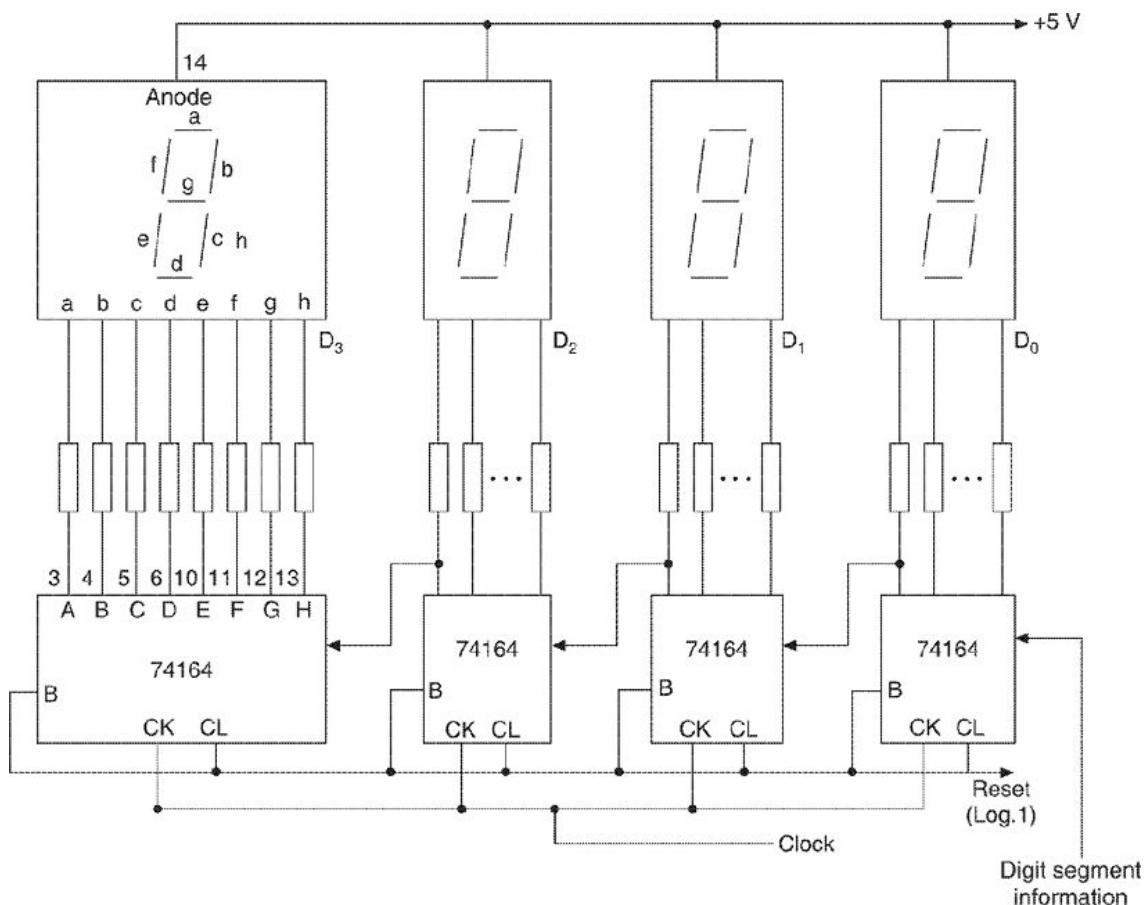


Figure 9.21 Serial interface of seven-segment LED display.

We input the MSB of the digit at data input and give a clock, then we input the next bit and again give a clock, so in total eight clock pulses,

we can display a digit. For the next digit, the same process is repeated and this way we can display as many number of digits as we want.

Two lines of any port can be used to input clock and digit segment information.

We determine the code of the digit to be displayed and store the codes of the digit in the consecutive memory locations. The code for D₃ will be stored first, then for D₂, and so on.

It is necessary to have Digit and Segment Counters. The maximum count for the Segment Counter is 8 and for the Digit Counter, it is 4.

After every bit/segment transfer, we decrement the Segment Counter and after every digit, the Digit Counter is decremented. As soon as both the counters become 0, the digit transfer is completed.

9.10 PROGRAMMING THE 8051 RESOURCES

The 8051 has two timer/counters, serial interface and a powerful interrupt facility. We shall see how these facilities can be programmed to suit the requirements of the applications.

9.10.1 Timer/Counters

The 8051 has two 16-bit timer/counters. These two timer/counters can be programmed independently. How does the 8051 know whether a timer/counter is functioning as a timer or as an event counter? The answer is simple. There is a bit in the TMOD SFR that specifies whether it is a timer or a counter. If this bit is set, the timer/counter will work as a counter; and if this bit is reset, the timer/counter will work as a timer.

Timer mode

When a timer/counter is functioning as a timer, the timer register (TH1 and/or TL1 for Timer 1 or TH0 and/or TL0 for Timer 0) is incremented after every machine cycle. That is, it will be working at 1/12th of the oscillator frequency because each machine cycle has got 12 oscillator periods.

As an example, after starting the spin motor in a washing machine, the next operation, i.e. the motor shut down, is performed after a fixed time interval.

Counter mode

If it is required to count the number of the occurrences of a particular event, the timer/counter needs to be used in the counter mode. Examples of applications include the axle counter system in railways to count the

number of wagons of a train passing a point and thus determining the length of the train; the counter in the elevator system to count the number of persons entering or exiting the elevator; and so on.

When a timer/counter is used as an event counter, the registers are incremented whenever a 1 to 0 transition is sensed at the T0 or T1 (depending on whether it is the Timer 0 or the Timer 1) pins of the 8051. The T0 and T1 pins are scanned during S5P2 of every machine cycle.

Let us take an example to make it more clear. Assume that in the first machine cycle, the T0 pin was high. The CPU will come to know that the T0 pin of Timer 0 was high during the S5P2 of the machine cycle. Now, in the second machine cycle, the CPU will again check for the T0 pin during S5P2. Let us assume now that the pin is low. So, the CPU comes to know that the transition from 1 to 0 has occurred, and hence it will increment the register during the S3P1 of the next machine cycle. Following are some conclusions that you can derive from the above example.

- The change in the state of the external input to T0 or T1 pin should hold for at least one machine cycle.
- The maximum count rate, that one can have (using external input at T0 or T1), is 1/24th of the oscillator frequency, as it needs at least two machine cycles to sense the changes.

Apart from the above mentioned pins and bits that control the operation of the timer/counters, there are some more bits and pins that have their effect on the timers. We will deal with them shortly. These timers can function in four different modes, namely mode 0, mode 1, mode 2 and mode 3. These modes are achieved by setting certain bits in the TMOD-register. Mode 0 to mode 2 are common for both the timers but not mode 3. These modes are described below in detail.

Mode 0: In this mode, TL0 and TH0 for Timer 0 (or TL1 and TH1 for Timer 1) are used as a 13-bit register, i.e. all the 8 bits of the TL0 or TL1 are utilized and the five lowermost bits of the TH0 or TH1 are used for counting purposes. As the count rolls over from all 1s in the register to all 0s, the interrupt flag is set. This timer interrupt flag is a bit, namely TF0 (for Timer 0) in the TCON, which is a special function register. From Figure 9.22, it is clear that if then the register is incremented after every machine cycle, i.e. at the rate of 1/12th of the oscillator frequency. Now, for the register to get incremented, the control switch shown in the

diagram should be closed. This switch is logic controlled. To close the switch, we must fulfill this logic. There are many ways by which this logic can be implemented. Some of them are given below for Timer 0.

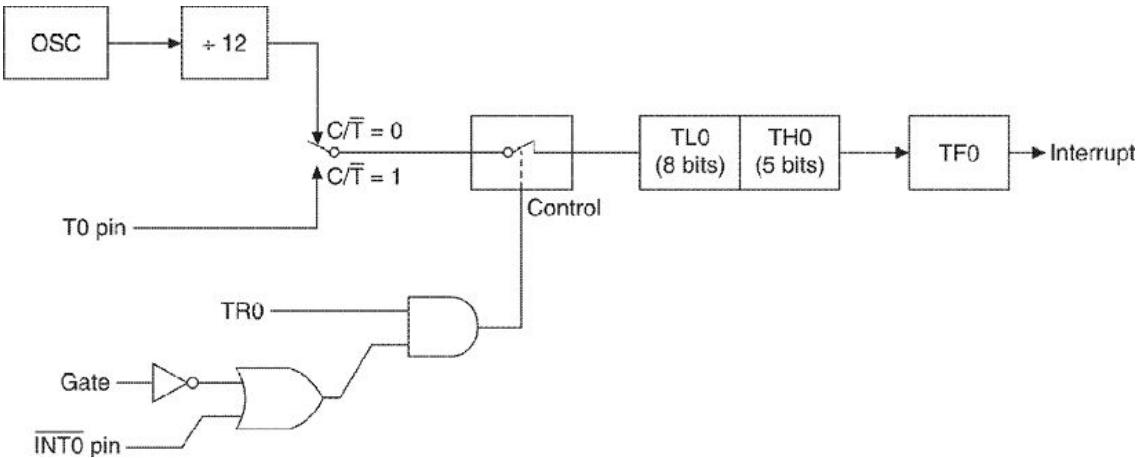


Figure 9.22 Timer 0, mode 0—13-bit counter.

- Case 1
TR0 = 1 (high)
Gate = 0 (low) and $\overline{\text{INT0}} = 0$ (low)
- Case 2
TR0 = 1 (high)
 $\overline{\text{INT0}} = 1$ (high) Gate = 1 (high)

Gate and TR0 are bits in the TMOD and TCON registers and can be set or reset through the software. $\overline{\text{INT0}}$ is one of the interrupt pins of the 8051.

In Case 2, the Gate is high. Hence, the incrementing of the register will depend on $\overline{\text{INT0}}$. This can be used for measuring the pulse width of a given signal. How? It is simple. Take Case 2 where TR0 is high and Gate is also high. Connect the signal, whose width is to be measured, to $\overline{\text{INT0}}$. Put the Timer 0 in the timer mode (by making $\text{C}/\bar{T} = 0$), i.e. it will work at 1/12th of the oscillator frequency. So, whenever the $\overline{\text{INT0}}$ pin goes high, the switch is closed and the counter will count at 1/12th of the oscillator frequency. This will continue till $\overline{\text{INT0}}$ becomes low. Now we can read the values of the register TL0 and TH0 and from that we can find the pulse width.

Mode 0 for both the timer/counters is the same, the only thing to do is to change TR0 to TR1, $\overline{\text{INT0}}$ to $\overline{\text{INT1}}$, and Gate bit for Timer 0 to Gate bit for Timer 1.

Mode 1: This mode is also the same for both the timers. This mode (Figure 9.23) is similar to mode 0, except that in this, 16 bits, that is, full TL0 and TH0 for Timer 0 (or TL1 and TH1 for Timer 1) are used for counting. So, the interrupt flag will be set only when the 16 bits go from all 1s to all 0s.

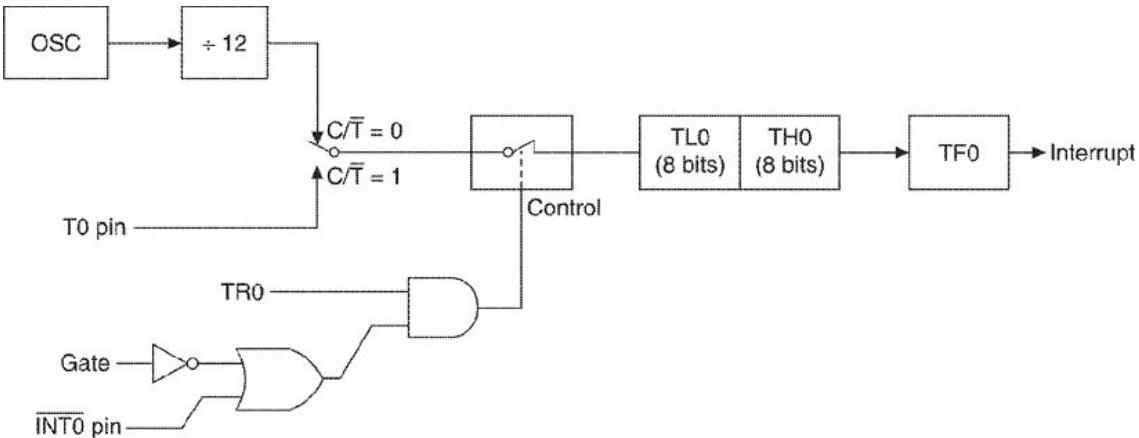


Figure 9.23 Timer 0, mode 1—16-bit counter.

Mode 2: In this mode, the timer register is 8 bits wide. TL0 for Timer 0 (or TL1 for Timer 1) is used for this purpose (Figure 9.24). This mode is also called the auto-reload mode as the timer generates an interrupt on overflow and after generating the interrupt, it also reloads the preset value from TH0 into TL0. This preset value can be put in the TH0 through a software. For example, let

$$TL0 = 80H$$

$$TH0 = 80H$$

Mode = 2

The interrupt flag is set when TL0 goes from all 1s to all 0s. After generating an interrupt, it also reloads the TL0 with the value from TH0 (80H in this case) and then starts counting again.

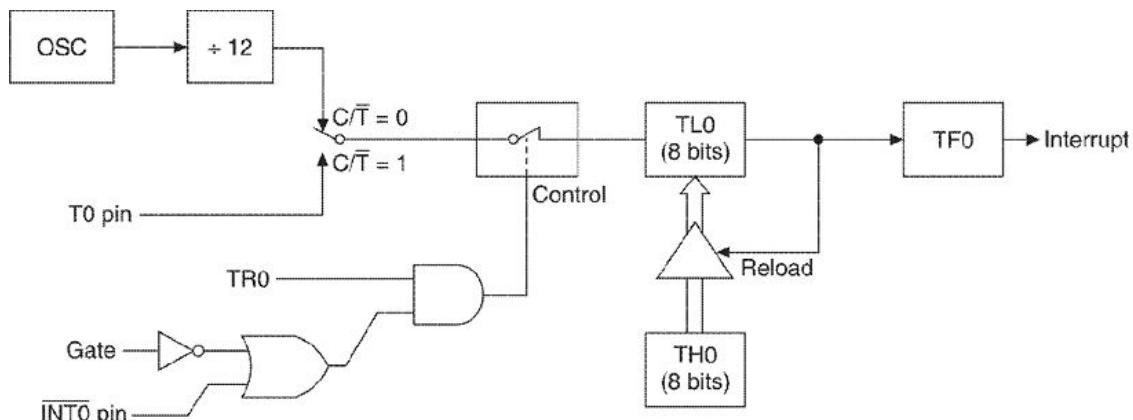


Figure 9.24 Timer 0, mode 2—autoreload.

Mode 3: If the Timer 0 is put into mode 3 (Figure 9.25), then it acts as two 8-bit counters (TL0 and TH0 become two separate counters). In this case, all the Timer 0 control bits (C/T , Gate, TR0, TF0 and $\overline{INT0}$) are used by TL0 itself and TH0 register is locked into a timer function. TH0 is counting machine cycles and has taken over the use of TR1 and TF1 from Timer 1. Therefore TH0 will now control Timer 1 interrupt. If the Timer 1 is put into mode 3, it just holds the count. The effect is same as setting TR1 = 0, hence opening the switch.

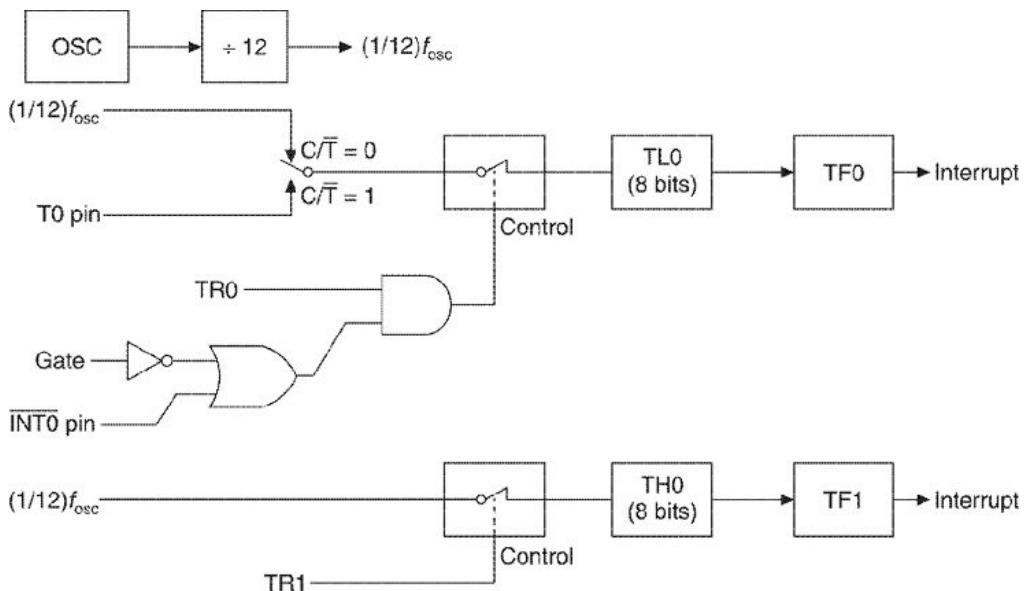


Figure 9.25 Timer 0, mode 3—split to two 8-bit counters.

Timer control and status register

The special function registers TMOD and TCON are used to control the timer/counter functions. When we write into these registers, the data is latched into them and takes effect at S1P1 of the next instruction (first cycle). The bits of these registers are described below:

TMOD: Timer Mode Control Registers

Timer 1				Timer 0				
Bit no.	7	6	5	4	3	2	1	0
Symbol	Gate	C/\bar{T}	M1	M0	Gate	C/\bar{T}	M1	M0

M1 and M0 specify the mode as follows:

M1	M0	Mode	Description in brief
0	0	0	13-bit counter
0	1	1	16-bit counter
1	0	2	8-bit counter with autoreload
1	1	3	Split Timer 0 into two 8-bit counters or to stop Timer 1

If $C/T = 1$, the timers function as counters to count the negative transitions at T0 or T1 pins.

If $C/T = 0$, the timers function as timers, that is, they basically count the number of machine cycles.

Gate = means that the timer is controlled by TR1 or TR0 only, irrespective of $\overline{INT0}$ or $\overline{INT1}$.

Gate = 1 means that the timer control will depend on $\overline{INT0}$ or $\overline{INT1}$ and also on TR0 or TR1 bits.

TCON: Timer Control Register

Bit no.	7	6	5	4	3	2	1	0
Symbol	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1: Timer 1 overflow flag. Set by hardware when the timer/counter overflows. Cleared by hardware when the processor vectors to the interrupt routine.

TR1: Timer 1 run control bit. Set/cleared by software to turn the timer/counter on/off.

TF0: Timer 0 overflow flag. Set by hardware when the timer/counter overflows. Cleared by hardware when the processor vectors to the interrupt routine.

TR0: Timer 0 run control bit. Set/cleared by software to turn the timer/counter on/off.

IE1: Interrupt 1 edge flag. Set by hardware when the external interrupt edge is detected. Cleared when the interrupt is processed.

IT1: Interrupt 1 type control bit. Set/cleared by software to specify the falling edge/low level triggered external interrupts. When IT1 = 1 then $\overline{INT1}$ is falling-edge triggered, otherwise when IT1 = 0 the $\overline{INT1}$ is low-level triggered.

IE0: Interrupt 0 edge flag. Set by hardware when the external interrupt edge is detected. Cleared when the interrupt is processed.

IT0: Interrupt 0 type control bit. Set/cleared by software to specify the falling edge/low level triggered external interrupts. When IT0 = 1 then $\overline{INT0}$ is falling-edge triggered, otherwise when IT0 = 0 the $\overline{INT0}$ is low-level triggered.

EXAMPLE 9.3

In the keyboard interface in the previous section, let us assume that the 8051 executes a program called KEYSCAN to scan the keyboard and keeps the data in memory. The program execution takes 10 milliseconds on a system with 12 MHz clock frequency. We need to regularly execute KEYSCAN in the interrupt mode, so that the system can utilize the time effectively.

Solution:

Clock frequency = 12 MHz

Count frequency = (1/12) clock frequency = 1 MHz

Time for 1 count = 1 μ s

Time for 2^{13} counts (mode 0) = 9.216 ms

Time for 2^{16} counts (mode 1) = 65.5 ms

Thus in case of mode 0, the time between two interrupts will be 9.216 ms, whereas in mode 1, it will be 65.536 ms. Depending on the application requirements, we may select one of the above two options.

Let us assume that we have selected mode 1. Then, KEYSCAN will be executed after every 65.536 ms. Let us also assume that we are using Timer 0 for this purpose.

Mode = 1, i.e. M1 M0 = 01

Timer Function, i.e. C/T = 0

Gate = 0

Thus, TMOD = 0000 0001 = 01H

It is also required to enable Timer 0 interrupt through IE SFR.

IE = 1000 0010 = 82H (refer Section 9.11.2)

Also Timer 0 must be initiated through TCON SFR

TCON = 00010000 = 10H

TCON:	EQU	88H
TMOD:	EQU	89H
IE:	EQU	A8H
	MOV	TMOD, #01H
	MOV	IE, #82H
	MOV	TCON, #10H

When Timer 0 overflows, i.e. from all 1s to all 0s transition, TF0 (Timer 0 overflow interrupt flag) will be set and the interrupt will be generated, i.e. the control will be transferred to the location 000BH. The user must store the KEYSCAN routine starting at 000BH, but should not overlap with the next Interrupt Servicing Routine address, i.e. 0013H for External Interrupt 1.

9.10.2 Serial Interface

The 8051 microcontroller chip has got all the circuitry in it for serial transmission. The RXD pin is used to receive the input serially and the TXD pin is used to transmit the data serially. The serial communication is full duplex, meaning that the 8051 can receive and transmit at the same time. The receiving unit is buffered as well. Thus, the reception of the second byte or the frame data can start even before the first byte is received by the CPU.

Since the buffer is only one byte wide, the CPU must read the previous byte before the second byte is fully received, otherwise one of the bytes will be lost. The special function register SBUF is used for both receiving and transmitting the byte. A write into the SBUF will initiate the process of transmission. A read will access a physically separate receive register. The serial port in the 8051 can be configured into four different modes, namely mode 0 to mode 3, depending on the application.

Mode 0: In this mode, the 8051 receives and transmits through the RXD pin. The TXD outputs the shift clock. 8 bits of data are received or transmitted. While transmitting, the LSB of the byte is sent out first. Similarly, while receiving, the LSB is received first. The baud rate in this mode is constant and it is 1/12th of the oscillator frequency. Figure 9.26 describes the function and the timing diagram associated with mode 0 serial interface. Once all the 8 bits of the data byte are transmitted, the Transmit Interrupt (TI) flag is set and an interrupt is generated. Similarly, after receiving all the 8 bits, the Receive Interrupt (RI) flag is set and an interrupt is generated. These TI and RI flags are nothing but the two bits in the SCON register. We shall later describe this register in detail.

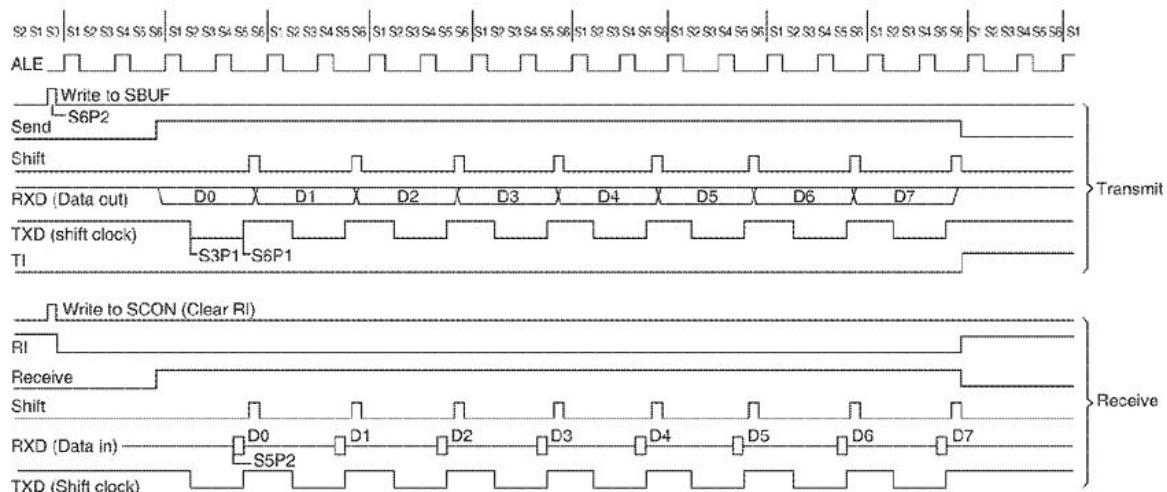
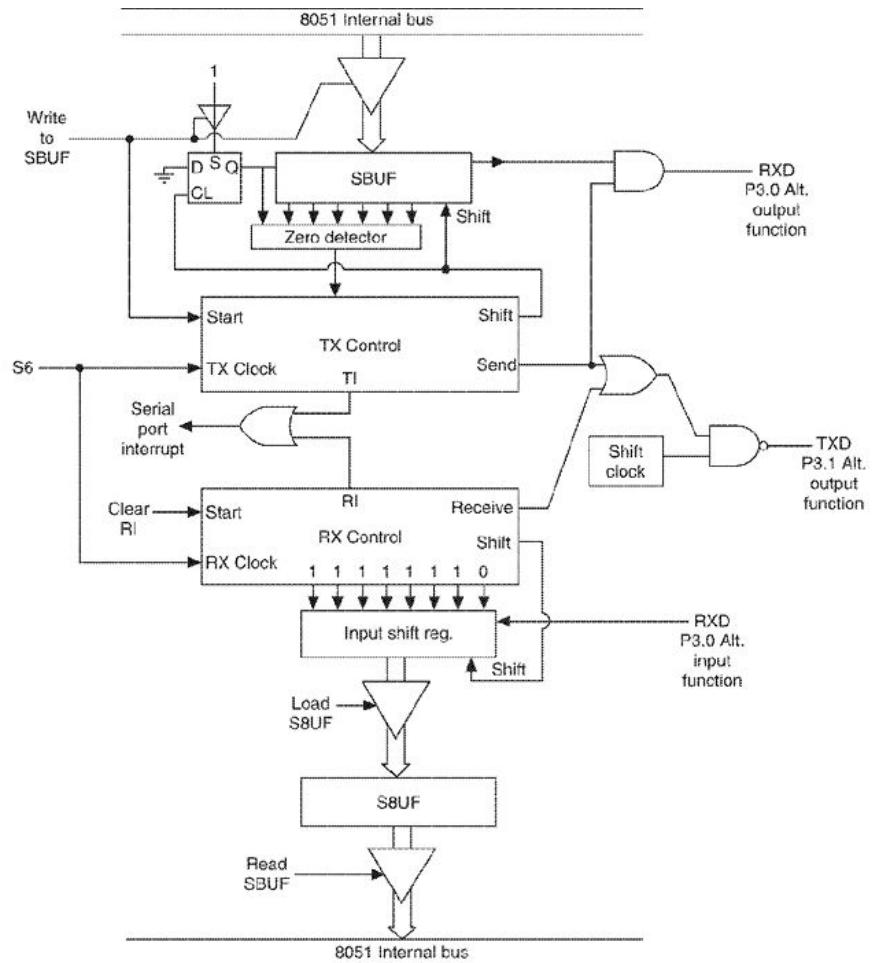


Figure 9.26 Serial interface—mode 0.

Mode 1: In this mode, the 8051 transmits 10 bits of information through TXD and receives 10 bits of information through RXD. The first bit is the start bit followed by the 8 bits of data (LSB first) and then a stop bit (high). Once the stop bit is received, it means that the reception of one frame is complete, that is, a byte of data has come. This stop bit is loaded into a bit called RB8 in the SCON register (Figure 9.27). As in mode 0,

here also an interrupt is generated once all the bits in a frame are received or transmitted. But unlike mode 0, here the baud rate is variable. The baud rate is set by the overflow rate of the Timer 1. The stop bit and the start bit are automatically added by the 8051 CPU through hardware while transmitting the data.

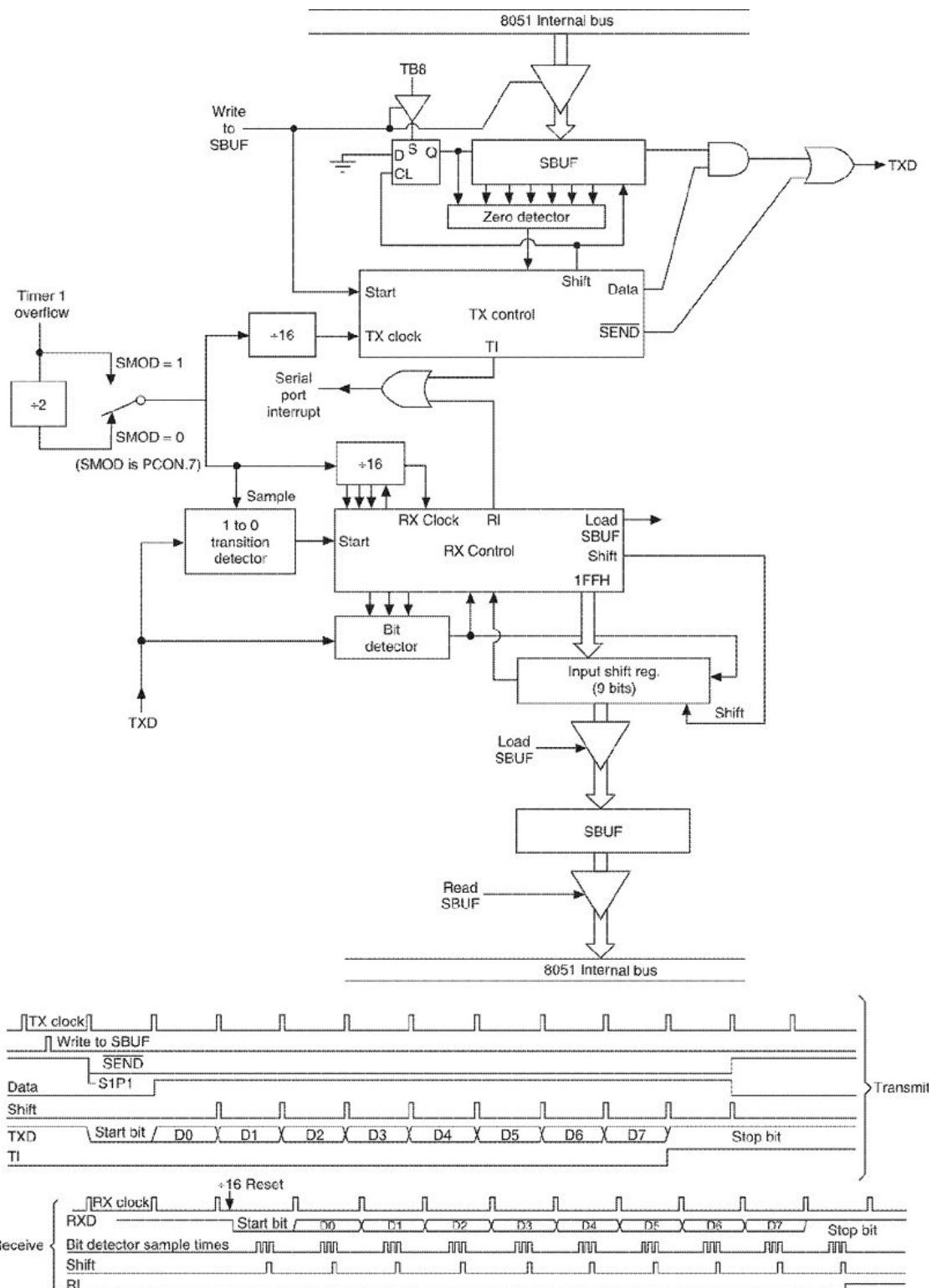


Figure 9.27 Serial interface—mode 1.

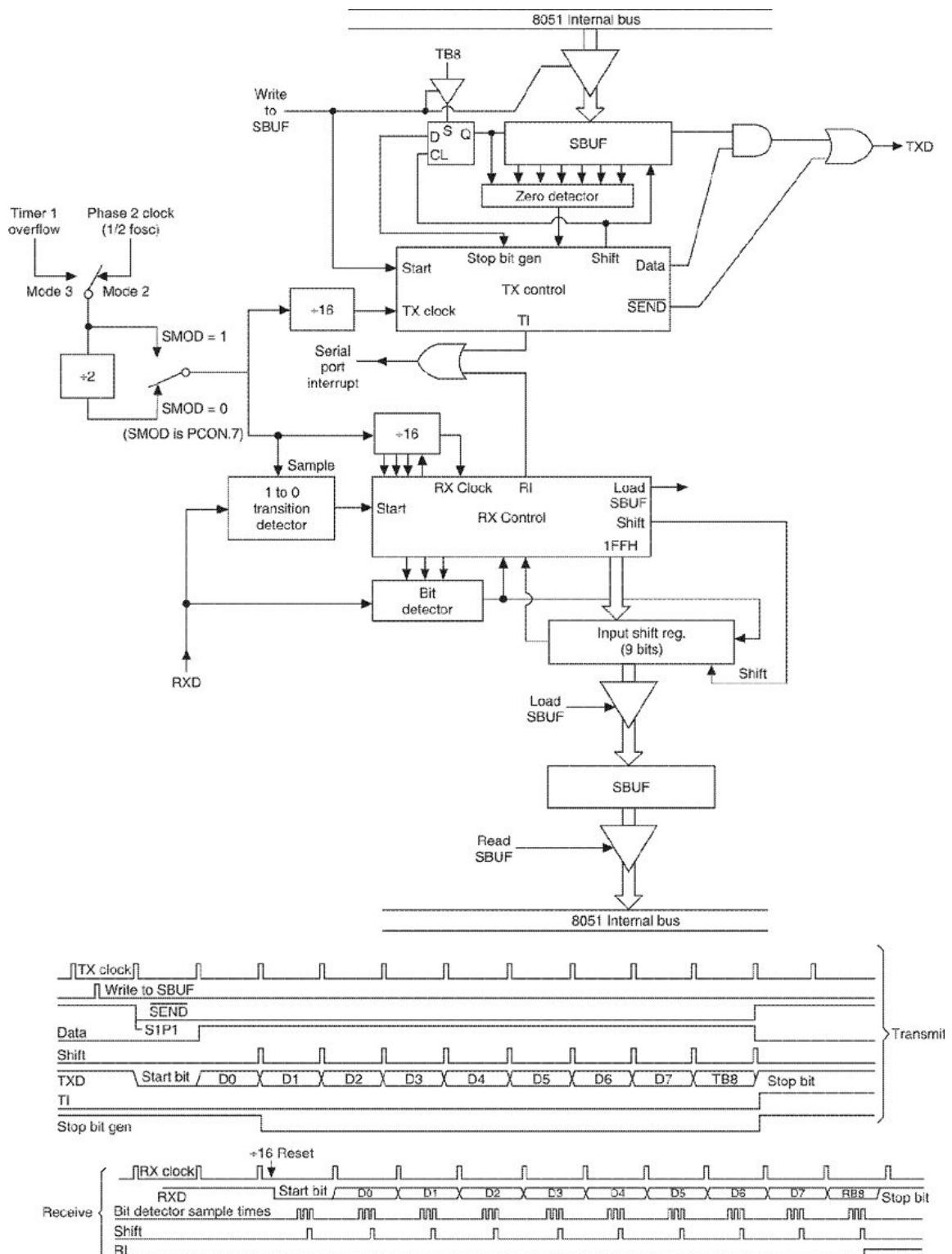


Figure 9.28 Serial interface—mode 2 and mode 3.

Mode 2: In this case, 11 bits are transmitted or received. This 11-bit frame is classified as shown below.

1. 1 bit for start
2. 8 bits for data

3. 1 bit can be programmed
4. 1 bit for stop

The 9th data bit is programmable. This 9th bit is nothing but the TB8 bit in the SCON register. This bit can be used effectively while transmitting the data, i.e. with little software overhead, this bit can be used to send the parity of the byte that is to be transmitted (Figure 9.28). Let us consider the two cases of transmitting and receiving separately.

Transmitting: If we want to use the 9th bit as parity, we must load the TB8 with the parity of the byte that is to be transmitted. ACC has the byte to be transmitted.

MOV C, P	Parity of ACC is loaded to carry
MOV TB8, C	Moves the parity which is in carry to TB8
MOV SBUF, A	Moves the byte into SBUF and initiates the transmission.

Receiving: On reception of a frame, the 9th bit goes to RB8 of the SCON register. The stop bit is ignored. The baud rate, in this mode, is programmable either 1/32th or 1/64th of the oscillator frequency.

Mode 3: This mode is same as the mode 2 except for the baud rate. Here the baud rate is variable. Timer 1 is used as the baud rate generator.

Serial control register (SCON)

This SFR controls the operations of the serial port. This register is used to define the operating modes. This also receives the 9th bit and contains the transmit and receive interrupt flags as well.

SCON : Serial Control Register

Bit no.	7	6	5	4	3	2	1	0
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0 and SM1 define the mode:

SM0	SM1	Mode	Description	Baud rate
0	0	0	Shift Register	1/12th Oscillator Frequency
0	1	1	8-bit UART	Variable
1	0	2	9-bit UART	1/64 Oscillator Frequency if SMOD = 0
				1/32 Oscillator Frequency if SMOD = 1
1	1	3	9-bit UART	Variable

SM2: Special feature for multiprocessor communication in mode 2 and mode 3. In mode 2 and mode 3, if SM2 = 1, the RI will not be activated unless the 9th bit received (RB8) is 1. In mode 1, if SM2 = 1, the RI will not be activated unless a valid stop bit is received. In mode 0, SM2 should be reset.

REN: Enables the serial reception. If set, it enables the reception, otherwise the reception is disabled.

TB8: It is the 9th bit of the data that is to be transmitted. Set and reset by software.

RB8: In modes 2 and 3, it is the 9th bit received. In mode 1, if SM2 = 0, RB8 is the stop bit that is received. In mode 0, it is not used.

TI: Set by hardware. This is known as the Transmit Interrupt flag. It is set at the end of the 8th bit in mode 0 and at the beginning of the stop bit in other modes, in any serial transmission. This has to be cleared by software.

RI: It is known as the Receive Interrupt flag, set by hardware at the end of the 8th bit in mode 0, or set at the reception of the stop bit in other modes. This must be cleared by software.

Any changes, if made in this SFR, get latched. This change is incorporated during the S1P1 of the first cycle of the next instructions.

Power control register (PCON)

It is a special function register through which certain power control functions in CMOS version of the 8051 are implemented. In the HMOS version, all the bits of PCON except bit 7 are dummy. Bit 7 is SMOD and is used in both CMOS and HMOS versions to double the baud rate in modes 1, 2 and 3. PCON is not bit addressable.

Baud rate

The baud rate, in mode 0, is fixed at 1/12th of the oscillator frequency. The baud rate in mode 2 is either 1/64th or 1/32th of the oscillator frequency depending on the bit value of SMOD in the special function register PCON. If SMOD = 0, the baud rate is 1/64th of the oscillator frequency. If SMOD = 1, the baud rate is 1/32th of the oscillator frequency.

For mode 1 and mode 3, the baud rates are variable. The baud rate is determined by the Timer 1 overflow rate, i.e.

$$\text{Baud rate} = \frac{\text{Timer 1 overflow rate}}{n}$$

where n is an integer and its value is either 32 or 16, i.e.

if $\text{SMOD} = 1$, then $n = 16$
and else $n = 32$.

In this case, Timer 1 can be configured in any mode. It is advisable to keep Timer 1 in mode 2 which is also the autoreload 8-bit counter. The timer is prevented from interrupting the processor by resetting the bit IE.3 in the IE register. Here, the overflow rate will depend upon the reload value in the TH1, i.e.

$$\text{Overflow rate} = \frac{\text{Count rate}}{256 \cdot \text{TH1}}$$

The timer can even be configured in other modes, but then we will have to enable the Timer 1 interrupt and also reload the timer counter register through software.

When $C/T = 0$, i.e. Timer 1 functioning as the timer, then the count rate = 1/12th of the oscillator frequency. If $C/T = 1$, the count rate is defined by the external input that is applied to the pin T1. In this case, the maximum count rate can be 1/24th of the oscillator frequency.

Table 9.2 shows the commonly used baud rates and how they can be obtained for Timer 1.

Table 9.2 Timer 1 configuration for different baud rates

Baud rate	Oscillator frequency	SMOD	Timer 1		
			Mode	Reload value	
Modes 1, 3 : 62.5K	12 MHz	1	0	2	FFH
19.2K	11.059 MHz	1	0	2	FDH
9.6K	11.059 MHz	0	0	2	FDH
4.8K	11.059 MHz	0	0	2	FAH
2.4K	11.059 MHz	0	0	2	F4H
1.2K	11.059 MHz	0	0	2	E8H
137.5	11.986 MHz	0	0	2	1DH
110	6 MHz	0	0	2	72H

EXAMPLE 9.4

Figure 9.29 shows two 8051-based systems X and Y, interfaced through serial port. Let us assume that the system X wishes to send 20 bytes to system Y in mode 0.

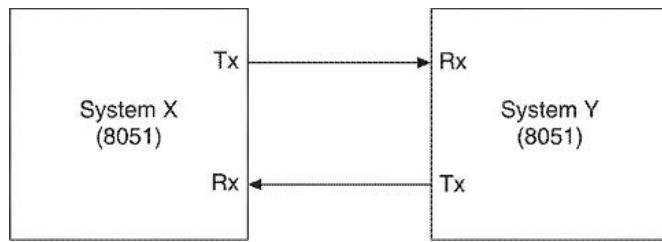


Figure 9.29 Serial communication between two systems.

Solution:

Now, the following will be the sequence of events in the two systems.

System X	System Y
Main Program	Main Program
$\text{SCON} = 00010000 = 10H$	$\text{SCON} = 00010000$
$= 10H$	
Enable serial port interrupt	Enable serial port
interrupt	
through IE SFR	through IE SFR
$\text{IE} = 10010000 = 90H$	$\text{IE} = 10010000 =$
$90H$	
Count $\square 20$	RI Interrupt
Load SBUF with the first byte	
..... Transmission starts	
:	
:	
: TI Interrupt	:
:	:
ISR for TI	ISR for RI
Clear TI flag	Clear RI flag
Decrement Count	Store SBUF in
memory	
If Count > 0 , then	Return
Load SBUF	
Return	
else Return	

9.10.3 Multiprocessor Communication

The bit SM2 in the SCON special function register helps in providing a simple protocol for multiprocessor communication. Mode 2 and mode 3 support this type of communication. In this, the 9th bit received, goes into the RB8 of SCON. The serial port can be programmed such that

only when the 9th bit (RB8) is a 1, the interrupt is generated. This feature can be invoked by setting the SM2 bit in SCON.

The steps that are involved in multiprocessor communication are given below.

1. When a processor in a multiprocessor environment has to send some data to another processor, it has to send the address of that processor on the communication bus.
2. Since this address will be received by all the processors on the network, every processor is going to get interrupted as soon as they receive all the bits of the address.
3. Each one of the processors will decode the address, and the addressed processor will only reply.
4. One aspect that has to be taken care of, is that only when an address is sent on the bus, the processors should get interrupted but not while the data is being sent. The data should only interrupt the processor whose address is sent first.

Now the obvious question is how is this achieved? There are two things one has to do for this. Firstly, the address bytes should have their 9th bit as a 1, so that RB8 is set and the data byte should have a 0 in the 9th bit. Secondly, SM2 should be initially set to a 1 in all the processors so that when the master sends an address on the bus, it should interrupt all the slaves.

Now the addressed slave will respond to the master by clearing its SM2 bit, so that the data that will be sent by the master now, can interrupt the slave that was addressed. At the end of the communication, the slave will again set the SM2 bit to a 1 so that it can now be interrupted only when an address is put on the bus by the master.

EXAMPLE 9.5

Figure 9.30 shows four 8051-based systems in master-slave configuration connected together through a serial port. The master has been designated DH as the hexadecimal address, whereas the slaves A, B and C have been designated AH, BH and CH as hexadecimal addresses. How will the operation be performed if the master system wishes to send 20 data bytes stored in memory to one of the slave systems, slave A, in mode 2.

Solution:

The operation will be performed in the following steps

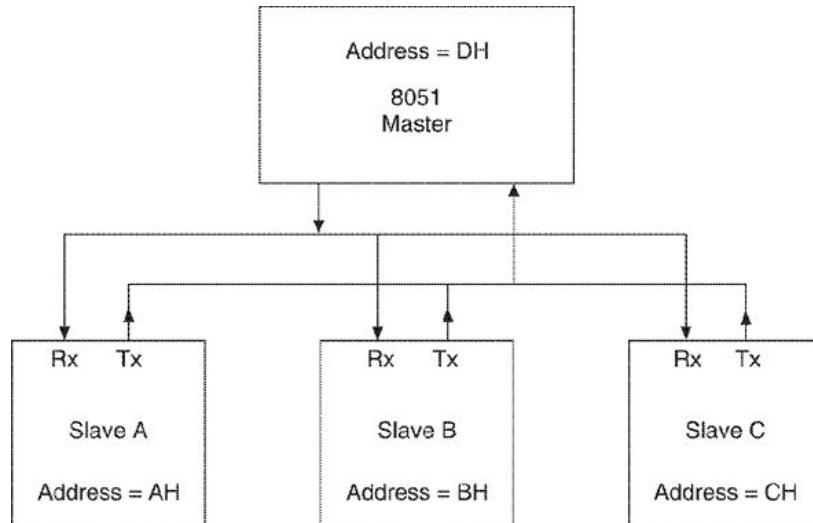
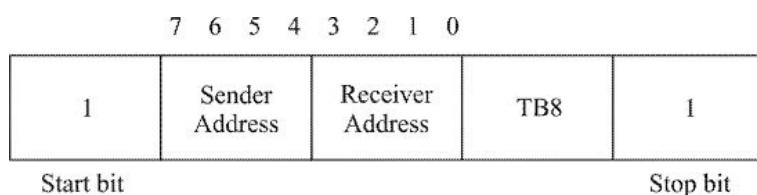
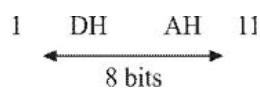


Figure 9.30 Multiprocessor communication.

- (a) In all the systems, master as well as slaves, serial port interrupt is enabled through IE SFR
 $IE = 10010000 = 90H$
- (b) SCON in the master system will be:
 $SM0, SM1 = 10, SM2 = 1, REN = 1, TB8 = 1, RB8 = 0, TI = 0, RI = 0$
i.e. $SCON = 10111000 = B8H$
- (c) SCON in the slave systems will be:
 $SM0, SM1 = 10, SM2 = 1, REN = 1, TB8 = 0, RB8 = 0, TI = 0, RI = 0$
i.e. $SCON = 10110000 = B0H$
- (d) The master sends the address frame. It is left to the designer to work out the handshaking protocol. Let us assume that the following address frame is adopted.



In our example, the address frame will be



- (e) All the slaves receive the address frame. RI interrupt is caused, since $RB8 = 1$.
RI ISR in slave systems
Clear RI Flag
If $SM2 = 0$, then Go To DATA CHECK

else compare the ‘Receiver Address’ with the System Address
 If Not Equal then Return
 else Reset SM2, Set TB8 = 1 (SCON = 10011000 = 98H), Send Ack Frame
 Return
 DATACHECK: Store Data byte in memory
 Return

(f) Let us consider the following as Ack Frame.

0	1 1 1 1	Sender Add.	TB8	0
Start bit				Stop bit

Master receives the Ack frame.

RI interrupt is caused since RB8 = 1

RI ISR in master

Reset SM2 (SM2 \square 0)

Reset TB8 (TB8 \square 0)

Fetch data byte from memory

Send data byte

Return

(g) The slave processor should get some indication at the end of the communication. It can be achieved in the following two ways.

1. The master can send the number of bytes to be transferred as the first byte. The slave may count the number of bytes received and at the end set SM2 = 1.
2. The master can send the byte whose code is pre-decided as the end of the message. The slave will check it and then set SM2 = 1.

9.11 INTERRUPTS

The 8051 has five interrupt sources. Each of these interrupts can be programmed to two priority levels. The interrupt sources are:

<i>Source</i>	<i>Description</i>
$\overline{\text{INT0}}$	External request from P3.2 pin.
Timer 0 TF0	Overflow from Timer 0 activates the Interrupt Request Flag
$\overline{\text{INT1}}$	External request from P3.3.
Timer 1 TF1.	Overflow from Timer 1 activates the Interrupt Request Flag
Serial port	Completion of the transmission or reception of a serial frame activates the flags TI or RI.

Each of these interrupt sources can be individually enabled or disabled by setting or clearing a bit in the Special Function Register IE. All these sources can be programmed either to a high priority level or to a low priority level, by setting or clearing the bits in IP (SFR). That is, if one of the sources is made as low priority, then it can be interrupted by another high priority interrupt, whereas a high priority interrupt cannot be interrupted by another high or low priority interrupt.

Now the obvious question that arises is, what will happen if two interrupts of the same priority come simultaneously? In this case, the interrupt priority is decided as given below:

<i>Source</i>	<i>Priority level</i>
External interrupt 0	(Highest)
Timer 0 overflow	
External interrupt 1	
Timer 1 overflow	
Serial port	(Lowest)

All these interrupts are separately scanned during each machine cycle and all the interrupts are prioritized by S6 of any machine cycle. The processor will go to the interrupt service in the first state of the next machine cycle, provided it is not blocked by any of the following conditions.

- An interrupt of equal or higher priority level is already in progress.
- The current machine cycle is not the final cycle in the execution of the instruction in progress, i.e. no interrupt request will be responded to until the instruction in progress is completed.
- The instruction in progress is RETI or an access to the Special Function Register IE or IP. In this case, the response will come only after executing at least one more instruction.

9.11.1 Response Time

The response time to an external interrupt is a very crucial factor in control applications where an immediate action is required. The response time will depend on the previously mentioned three cases. Let us take the first case. In this, if an interrupt of low priority comes and the processor is already in a higher priority interrupt subroutine, then the response time will depend on the nature of the interrupt service routine that is being executed.

In the second case, the response time will be at the most three machine cycles, as the longest instructions are MUL and DIV which take four machine cycles.

In the third case, under normal circumstances, the delay in response can be at the most five machine cycles, i.e. a maximum of one more machine cycle to complete the first instruction which may be RETI or access to IE or IP, and then if the next instruction is MUL or DIV it will take four more machine cycles for the execution.

Once an interrupt is not blocked by any of the above mentioned cases, then a hardware call is generated internally by the processor, after which the program branches to a predefined location. The vector addresses for the various sources are given below.

<i>Source</i>	<i>Location</i>
	0003H
T0	000BH
	0013H
T1	001BH
Serial Port	0023H

Before going to these locations, it stores the program counter value in the stack, so that when RETI (Return from Interrupt) instruction of the Interrupt Service Routine (ISR) is executed, the processor can return to the location where the interrupt took place.

9.11.2 Interrupt Control Registers

The interrupt request flags are in two different registers and two port pins, as listed below.

<i>Source</i>	<i>Request flag</i>	<i>Location</i>
External	, if IT0 = 0	P3.2
Interrupt 0	IE0, if IT0 = 1	TCON.1
Timer 0	TF0	TCON.5
Overflow		
External	, if IT1 = 0	P3.3
Interrupt 1	IE1, if IT1 = 1	TCON.3
Timer 1	TF1	TCON.7
Overflow		
Serial port	TI (On transmission)	SCON.1
	RI (On reception)	SCON.0

External Interrupt control bits IT0 and IT1 are in TCON.0 and TCON.2 respectively. Reset leaves all the flags inactive, with IT0 and IT1 cleared. All the interrupt flags can be set or cleared by software, with the same effect as by hardware.

External interrupts can be programmed to be either level triggered or edge triggered by making the bit ITx in TCON as a 0 or a 1. In case of ITx = 0, i.e. level-triggered activation, a low-level signal at $\overline{\text{INT}0}$ or $\overline{\text{INT}1}$ will cause the interrupt. On the other hand, when ITx = 1, i.e. edge-triggered activation, a high-to-low transition of the signal at $\overline{\text{INT}0}$ or $\overline{\text{INT}1}$ will cause an interrupt. In short, if

IT0, i.e TCON.0 = 0, $\overline{\text{INT}0}$ is low-level triggered

IT0, i.e TCON.0 = 1, $\overline{\text{INT}0}$ is high-to low level edge triggered

IT1, i.e TCON.2 = 0, $\overline{\text{INT}1}$ is low-level triggered

IT1, i.e TCON.2 = 1, $\overline{\text{INT}1}$ is high-to-low level edge triggered

The Interrupt Enable and Priority Control Registers are shown below. All of these control bits are set or cleared by software. All are cleared by reset.

IE: Interrupt enable register

7	6	5	4	3	2	1	0
EA	X	X	ES	ET1	EX1	ET0	EX0

where

EA Disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

ES Enables or disables the Serial Port interrupt. If ES = 0, the Serial Port interrupt is disabled.

ET1 Enables or disables the Timer 1 Overflow interrupt. If ET1 = 0, the Timer 1 interrupt is disabled.

EX1 Enables or disables the External Interrupt 1. If EX1 = 0, the External Interrupt 1 is disabled.

ET0 Enables or disables the Timer 0 Overflow interrupt. If ET0 = 0, the Timer 0 interrupt is disabled.

EX0 Enables or disables the External Interrupt 0. If EX0 = 0, the External Interrupt 0 is disabled.

IP: Interrupt priority register

7	6	5	4	3	2	1	0

X	X	X	PS	PT1	PX1	PT0	PX0
---	---	---	----	-----	-----	-----	-----

where

PS Serial Port interrupt priority level. PS = 1 programs it to the higher priority level.

PT1 Timer 1 interrupt priority level. PT1 = 1 programs it to the higher priority level.

PX1 External Interrupt 1 priority level. PX1 = 1 programs it to the higher priority level.

PT0 Timer 0 interrupt priority level. PT0 = 1 programs it to the higher priority level.

PX0 External Interrupt 0 priority level. PX0 = 1 programs it to the higher priority level.

EXAMPLE 9.6

In Figure 9.31, the 8051 system is used to maintain the power supply voltage to another system, which may also be a complete process plant. As shown in the schematic, the supply voltage monitoring subsystem continuously checks the conditions of mains supply, both voltage and frequency. It sends interrupt signals on INT0 line, if the mains supply condition deteriorates below a specified level. The 8051 system, on receipt of INT0, switches on the backup supply through Relay 1. When the mains supply condition improves to the specified level, the supply voltage monitoring subsystem sends interrupt INT1 and the 8051 system switches on the mains supply. Design the system.

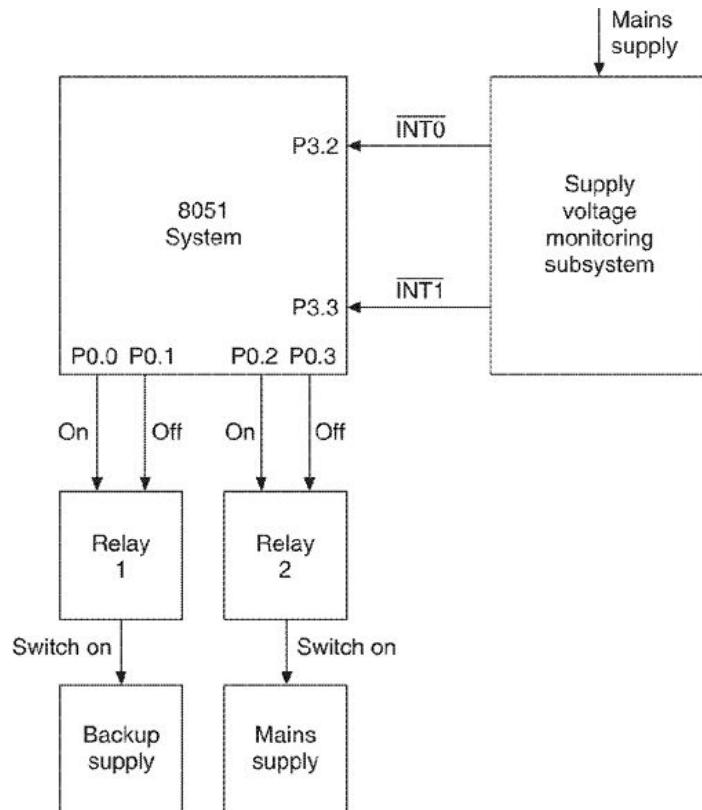


Figure 9.31 The 8051 interrupt system: an example.

Solution:

(a) The first task that a designer needs to perform, is to find out the type of interrupts (level/edge triggered) and set the TCON special function register. Let us assume that both the interrupts are low-level triggered. Thus,

$$\text{IT0, i.e. TCON.0} = 0$$

$$\text{IT1, i.e. TCON.2} = 0$$

If the timer/counter is not being used, then

$$\text{TCON} = 00\text{H}$$

(b) The interrupts need to be enabled, through IE SFR. Thus,

$$\text{IE.0} = \text{EX0} = 1, \text{IE.2} = \text{EX1} = 1$$

$$\text{IE.7} = \text{EA} = 1$$

$$\begin{aligned} \text{Thus, IE} &= 1000\ 0101 \\ &= 85\text{H} \end{aligned}$$

(c) The interrupt priorities need to be programmed through IP SFR. Let us allocate both the interrupts to high priority level. Thus,

$$\text{IP.0} = \text{PX0} = 1$$

$$\text{IP.2} = \text{PX1} = 1$$

Thus $IP = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
= 05H

- (d) Having decided the contents of SFRS, TCON, IE and IP, the designer can load the values through MOV instructions in the main program.
- (e) The Interrupt Service Routine (ISR) for $\overline{\text{INT0}}$ needs to be located at location 0003H, whereas ISR for $\overline{\text{INT1}}$ should be located at 0013H. The algorithms will be

ISR- $\overline{\text{INT0}}$	1. Switch off mains by setting P0.3 = 1 and P0.2 = 0. 2. Switch on backup by setting P0.0 = 1 and P0.1 = 0. 3. Enable Interrupt by setting IE.7 = 1. 4. Return.
ISR- $\overline{\text{INT1}}$	1. Switch on mains by setting P0.2 = 1 and P0.3 = 0. 2. Switch off backup by setting P0.1 = 1 and P0.0 = 0. 3. Enable Interrupt by setting IE.7 = 1. 4. Return.

9.12 MEASUREMENT OF FREQUENCY, PERIOD AND PULSE WIDTH OF A SIGNAL

There are many applications which require the measurement of frequency, period and pulse width of a signal. As an example, pulse width modulated (PWM) signals with different duty cycles may be used for controlling the motors. The control information is encoded in the signal in pulse width and duty cycle. The measurement using resonant transducers is another important application. Resonant transducers are oscillators whose frequencies depend, in a known way, on the physical property being measured. These devices output a train of rectangular pulses whose repetition rate encodes the value of the quantity being measured.

The source, generating the signal, may be directly connected to the 8051. The accuracy of measurement will depend on the accuracy of the clock oscillator of the 8051. The signal source may be directly connected to either the interrupt ($\overline{\text{INT0}}$ or $\overline{\text{INT1}}$) pins or the timer/counter pins (T0 or T1) as shown in Figure 9.32. These are the alternate function pins of Port 3. Now we shall take up the measurements of frequency, period and pulse width of the signal stream.

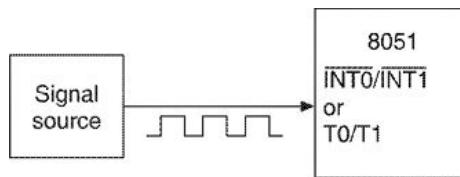


Figure 9.32 Measurement of frequency, period and pulse width using the 8051.

9.12.1 Frequency Measurement

The measurement of frequency requires

- (a) measurement of sample time, and
- (b) counting of pulses during the measured sample time.

This can be achieved using two timer/counters. One timer/counter is used to measure the time whereas, another timer/counter is used to count the number of pulses. Let us assume that T0 is used for time measurement and T1 is used for pulse counting. Let us also assume that the clock frequency of the 8051 is 12 MHz. Now, the following operations will be performed.

Timer operation (Timer/Counter 0)

Clock frequency = 12 MHz

Count frequency = (1/12) clock frequency = 1 MHz

Time for 1 count = 1 μ s

Time for 2^{13} counts (mode 0) = 9.216 ms

Time for 2^{16} counts (mode 1) = 65.5 ms

Thus, in case of mode 0, the time between two interrupts will be 9.216 ms, whereas in mode 1, it will be 65.536 ms. Any one of the two options may be selected.

Counter operation (Timer/Counter 1)

As mentioned earlier, it takes two machine cycles to sense the change. Thus, the maximum count rate will be 1/24th of the oscillator frequency. For 12 MHz clock frequency, the count frequency that can be measured will be 500 kHz.

Maximum total count in 9.216 ms = 4608

Maximum total count in 65.5 ms = 32750

These numbers can be represented in a 16-bit count register (TH1 TL1), and there will be no need to augment this with the software counter. The operation sequence will be

- (i) Program Timer/Counter Mode

Timer/Counter 0

Mode = 0, i.e. M1M0 = 00 (13-bit operation)

Timer function, i.e. $C/\bar{T} = 0$

Gate = 0

Timer/counter 1

Mode = 1, i.e. M1 M0 = 01 (16-bit operation)

Counter function, i.e. $C/\bar{T} = 1$

Gate = 0

Thus, TMOD = 0 1 0 1 0 0 0 0 = 50H

MOV TMOD , #50H

(ii) Initiate Timer/Counter

The timer/counters may be initiated by making TR0 and TR1 bits in TCON register as 1. Thus

TCON = 0 1 0 1 0 0 0 0 = 50H

It is also required to enable Timer 0 interrupt through IE register.

IE = 1 0 0 0 0 0 1 0 = 82H

MOV IE, #82H

MOV TCON, #50H

(iii) Calculate Frequency

When Timer 0 reaches its final count and then becomes 0 (i.e. 2^{13} count of the oscillator clock in mode 0), it will generate the Timer 0 interrupt. If there is no higher priority interrupt pending, then ISR at location 000BH will be executed. The ISR should read the count value in TH1 TL1 and then calculate the frequency based on the count.

$$\text{Frequency} = \frac{\text{Count}}{\text{Sample Time}} = \frac{\text{Count}}{9.216 \text{ ms}}$$

It is also possible to use the look-up table method in which counts and the corresponding frequency values are stored in memory, and are readily accessed.

There is another approach for the measurement of frequency, which uses only one timer/counter. The signal source in this case is connected to one of the external interrupt inputs INT0 or INT1. The ISR increments a software counter when executed. Thus, Timer/Counter 1 which performs the pulse count in the earlier method is not required. The ISR for T0 interrupt will read the software counter and perform the calculation.

In this case, the 8051 may be performing other tasks in addition to frequency measurement, or frequency measurement may be a part of an application. However, due to response time constraints of external interrupts mentioned earlier, a delay of five machine cycles may occur in interrupt response. Moreover, it takes two machine cycles to recognize and prioritize the interrupts. In addition, the execution of the instruction in ISR to maintain the software counter, branching to ISR and returning from ISR, pushing and popping of return address, will also delay the response time. Roughly, the overall delay may amount to 20 machine cycles and thus the maximum frequency of the signal, in this case may be around 50 kHz.

If the signal frequency to be measured falls within the known frequency limits of f_{\max} and f_{\min} , then the measurement can be fine tuned between these two frequencies to derive a better resolution.

The full-scale range of the count will be $T \square (f_{\max} - f_{\min})$. If the count is represented in n bits, the full-scale range of count will be $= 2^n = T \square (f_{\max} - f_{\min})$

$$\text{LSB of count} = T \square (f_{\max} - f_{\min})/2^n$$

$$\text{Sample time required } T = 2^n/(f_{\max} - f_{\min})$$

As an example, let us suppose that the frequency to be measured falls between the known range of 10 kHz and 14 kHz. For 8-bit resolution, the sample time will be $256/4 = 64$ ms. The count for f_{\max} will be $64 \square 14 = 896$, whereas the count for f_{\min} will be $64 \square 10 = 640$.

Thus, the count will vary between 640 and 896. By presetting the counter to 640, we will get pulse count between 00 and 256 (00H and FFH) corresponding to f_{\min} to f_{\max} frequency range for sample time of 64 ms. The count may be divided by 64 to get the frequency relative to f_{\min} . The f_{\min} may be added to this relative frequency to get the actual signal frequency.

As an example, consider that the pulse count of 192 has been obtained after presetting the counter to 640. Thus, the frequency relative to f_{\min} will be $= 192/64 = 3$ kHz. Actual signal frequency $= f_{\min} + 3 = 10 + 3 = 13$ kHz.

However, it needs to be ensured that the timer/counter, performing the time measurement, interrupts the 8051 on reaching the sample time. The 8051 then reads the pulse count and calculates the frequency.

The timer register may be preset to a value, so that it rolls over on sample time and the interrupt is caused. The required preset value is represented in the form of sample time measured in machine cycles. If the sample time measured in machine cycles can be represented in 16 bits, then the preset value is FFFFH—sample time in machine cycles + 1. The timer register is incremented after every machine cycle and thus it will roll over to zero on the sample time and an interrupt will be caused.

In our example, the sample time is 64 ms and the clock frequency is 12 MHz. Thus, a machine cycle is of 1 μ s duration and the total number of machine cycles in sample time of 64 ms will be 64000, i.e. FA00H. The required preset value for the timer register is FFFFH – FA00H + 1 = 0600H.

If the sample time measured in machine cycles is more than 65536, the hardware timer on chip will need to be augmented by the software timer. As an example, if the sample time in machine cycles is 96000, i.e. 017700H, requiring 3 bytes, then 01H will be stored in the software timer and 8900H (FFFFH – 7700H + 1) will be loaded to hardware timer/counter. When the timer is initiated, it will cause interrupt when the count rolls over to 0000 from FFFFH. The ISR must check the software counter.

- If the software counter = 0, then the ISR must return to the main program without doing anything.
- If the software counter > 0, then the ISR must decrement the counter, load 0000H in the hardware counter, initiate the counter, enable timer interrupt and then return. When the timer count rolls over to 0000 from FFFFH (i.e. total timer count of 010000H), it will cause the interrupt.

Following is the sequence of operations.

- (i) Determine the sample time required from frequency limits (f_{\max} , f_{\min}) and the number of bits (n) available for storage.

$$\text{Sample time } T = 2^n / (f_{\max} - f_{\min})$$

In our example, $f_{\min} = 10 \text{ kHz}$, $f_{\max} = 14 \text{ kHz}$

$n = 8$ and thus $T = 64 \text{ ms}$

- (ii) Determine the count corresponding to f_{\max} and f_{\min} by multiplying the sample time. In our example, count for $f_{\max} = 64$

$\square 14 = 896$ and the count for $f_{\min} = 64 \square 10 = 640$.

(iii) Program timer/counter mode. Let us assume that Timer/Counter 0 is used for time measurement and Timer/Counter 1 for pulse counting.

Timer/Counter 0

Mode = 1, i.e. M1 M0 = 01 (16-bit operation)

Timer function, i.e. $C/\bar{T} = 0$

Gate = 0

Timer/Counter 1

Mode = 1, i.e. M1 M0 = 01 (16-bit operation)

Counter function, i.e. $C/\bar{T} = 1$

Gate = 0

Thus, TMOD = 0 1 0 1 0 0 0 1 = 51H

MOV TMOD, #51H

(iv) Load the preset value to the counter register. The count for f_{\min} , i.e. $T \cdot f_{\min}$ has been calculated in step (ii). The counter register is loaded with $T \cdot f_{\min}$ as the preset value. In our example, Timer/Counter 1 is loaded with preset value of $640 = 280H$. The value loaded in the Timer/Counter 1 = FFFFH – 280 + 1 = FD80H

MOV TL1, #80H

MOV TH1, #FDH

(v) Determine the sample time in machine cycles by multiplying the sample time with machine cycle duration (i.e. $T \square$ m/c cycle duration). In our example, since the clock frequency is assumed as 12 MHz, each machine cycle has the duration of 1 \square s. Thus the sample time in machine cycles = $64000 = FA00H$.

(vi) Calculate and load the preset value for timer. If the sample time in machine cycles occupies three bytes (i.e. if it is > 65536), then the existing hardware timer has to be augmented by the software timer and the preset value for the software timer also needs to be calculated and loaded.

(a) If the sample time in machine cycles occupies only two bytes, then

Preset value = FFFFH – (Sample time in machine cycles) + 1

In our example:

Preset value = FFFFH – FA00H + 1 = 0600H

MOV TL0, #00H

MOV TH0, #06H

(b) If the sample time in machine cycles occupies three bytes, then:

- Calculate and load the preset value in the timer register taking the lower two bytes of the sample time in machine cycles in the same way as in (a) above.

Preset value for timer register = FFFFH – (value in lower two bytes of sample time in machine cycles) + 1

- Load the third byte in software counter. For example, if the sample time in machine cycles is 96000, i.e. 017700H, then:

Preset value for timer register = FFFFH – 7700H + 1 = 8900H

```
MOV TL0, #00H  
MOV TH0, #89H  
MOV SCOUNT, #01H
```

where SCOUNT variable represents the software counter.

(vii) Initiate Timer/Counter.

The timer interrupt for Timer/Counter 0 needs to be enabled through IE register

IE = 1 0 0 0 0 0 1 0 = 82H

The timer/counters may be initiated by making TR0 and TR1 bits in TCON register as 1.

TCON = 0 1 0 1 0 0 0 0 = 50H

MOV IE, #82H

MOV TCON, #50H

(viii) Service Timer Interrupt

When the Timer 0 count in TH0 TL0 reaches FFFFH and rolls over to 0000, it will cause the Timer 0 interrupt and the ISR at location 000BH will be executed.

(a) If the software counter > 0, then the ISR must decrement the counter, load 0000H in the hardware counter TH0, TL0, initiate the counter through TCON, enable the timer interrupt through IE and return.

In our example:

```
DEC SCOUNT; Decrement software counter  
MOV TL0, #00H  
MOV TH0, #00H  
MOV IE, #82H  
MOV TCON, #50H
```

RETI

- (b) If the software counter = 0, then the ISR returns to the main program for frequency calculation.
- (ix) To calculate the frequency, the pulse count in the counter register must be read. The count value must be divided by T (i.e. sample time) to get the frequency relative to f_{min} . The relative frequency is added to f_{min} to get the actual frequency.

MOV A, TL1

; Due to presetting, the count will occupy only 8 bits

MOV B, T; T = Sample time

DIV AB

; ACC will contain the quotient, i.e. integer part of relative frequency

; B will contain the remainder

ADD A, f_{min}

In our example, if the pulse count = 192 then TL1 = C0H

Since T = 64 ms, i.e. 40H, the frequency relative to f_{min} = 03

Since f_{min} = 10 kHz, Signal Frequency = 13 kHz

9.12.2 Period Measurement

The measurement of the period of a signal involves the measurement of the total elapsed time over a known number of pulses. The period measurement is similar to frequency measurement and will require two timer/counters—one for counting the pulses, and the other for time measurement. Let us assume that Timer/Counter 0 is used for pulse counting. We may select the total number of pulses N , over which time measurement will be required and then preset the timer/counter to a value = FFFF – N + 1. The timer/counter will generate interrupt after N pulse counts.

For time measurement, let us assume that Timer/Counter 1 is used with 12 MHz as clock frequency. The operation sequence will be:

- (i) Program Timer/Counter mode

Timer/Counter 0

Mode = 01, i.e. M1 M0 = 01

Counter function, i.e. $C/T = 1$

Gate = 0

Timer/Counter 1

Mode = 01, i.e. M1 M0 = 01

Timer function, i.e. $C/T = 0$

Gate = 0

Thus, TMOD = 0 0 0 1 0 1 0 1 = 15H

MOV TMOD, #15H

- (ii) Store the preset count value in Timer/Counter 0. Let us assume that count of 1000 pulses is required to measure the period. Thus, the counter must be preset to value

FFFFH – 03E8H + 1 = FC18H

MOV TL0 #18H

MOV TH0, #FCH

- (iii) Initiate Timer/Counter. The timer/counter may be initiated by making TR0 and TR1 bits in TCON register as 1. Also the Timer 0 interrupt must be enabled through the IE register.

IE = 1 0 0 0 0 0 1 0 = 82H

TCON = 0 1 0 1 0 0 0 0 = 50H

MOV IE, #82H

MOV TCON, #50H

- (iv) Calculate Period

When Timer/Counter 0 interrupt occurs on 1000 counts, the ISR at 000BH location will be executed.

- (a) The ISR should read the time values from TH1 TL1. For 12 MHz clock frequency, each count in TH1 TL1 represents 1 μ s.
- (b) Calculate the period by dividing time by pulse count value, i.e. 1000 in our example. If the time value in TH1 and TL1 is k ms, the period will be $k \mu$ s. Thus, no division will be required.

If the frequency of the signal, whose period is to be measured, falls within the known limits of f_{\max} and f_{\min} , then the measurement can be fine-tuned to derive better resolution. The relationship between period T of signal and its frequency f is

$$T = (f_{\text{xtal}}/f) \square (1/12)$$

where f_{xtal} is the clock frequency in the same unit as signal frequency f . If the signal frequency falls between f_{\max} and f_{\min} , then the period T will also fall between the limits T_{\max} and T_{\min} defined as

$$T_{\min} = (f_{\text{xtal}}/f_{\max}) \square (1/12)$$

$$T_{\max} = (f_{\text{xtal}}/f_{\min}) \square (1/12)$$

The full-scale range of the period will be $N \square (T_{\max} - T_{\min})$, where $N = \text{no. of pulses of the signal selected for measurement}$. If the period is represented in n bits, then the full-scale range of period $= 2^n = N(T_{\max} - T_{\min})$.

$$\text{LSB of period} = N(T_{\max} - T_{\min})/2^n$$

Thus the number of periods over which the elapsed time should be measured, is

$$N = 2^n/(T_{\max} - T_{\min})$$

The T_{\max} and T_{\min} must be in the machine cycle and N must be an integer. If the above formula results in a number with the decimal point, then the next higher integer value must be selected for N .

As an example, if $f_{\min} = 10 \text{ kHz}$ and $f_{\max} = 14 \text{ kHz}$ and clock frequency $f_{\text{xtal}} = 12 \text{ MHz}$, then:

$$T_{\max} = (12000/10) \square (1/12) = 100$$

$$T_{\min} = (12000/14) \square (1/12) = 71$$

For 8-bit resolution ($n = 8$)

$$N = 256/(100 - 71) = 256/29 = 8.8$$

The next higher value of integer, i.e. 9 is selected for N . Thus:

$$\text{Maximum value of machine cycles} = N \square T_{\max} = 900$$

$$\text{Minimum value of machine cycles} = N \square T_{\min} = 639$$

A look-up table may be created for the complete range of $N \square T$ and the corresponding value of T in 8 bits, and thus T can be easily determined.

Following will be the sequence of operations:

- (a) Determine T_{\max} and T_{\min} from the values of f_{\max} , f_{\min} and clock frequency.
- (b) Based on the value of n , determine the value of N (i.e. the number of signal pulses to be sampled), $N.T_{\max}$ and $N.T_{\min}$.
- (c) Create look-up table in memory from $N.T_{\min}$ to $N.T_{\max}$ and the corresponding value of T .
- (d) Program Timer/Counter mode
Timer/Counter 0 (used for pulse counting)

Mode = 01, i.e. M1 M0 = 01

Counter function, i.e. $C/T = 1$

Gate = 0

Timer/Counter 1 (used for time measurement)

Mode = 01, i.e. M1 M0 = 01

Timer function, i.e. $C/T = 0$

Gate = 0

Thus, TMOD = 0001 01 01 = 15H

MOV TMOD, #15H

(e) Load the preset count value in Timer/Counter 0. The preset value to be loaded to Timer/Counter 0 will be $N \cdot T_{\min}$.

In our example, $N \cdot T_{\min} = 639$

Preset value of 639 (= 027FH), i.e. FFFFH – 027FH + 1 = FD81H must be loaded

MOV TL0, #81H

MOV TH0, #FDH

(f) Initiate Timer/Counter

The Timer 0 interrupt must be enabled through IE. The Timer/Counter may be initiated using the TCON register.

IE = 1 0 0 0 0 0 1 0 = 82H

TCON = 0 1 0 1 0 0 0 0 = 50H

MOV IE, #82H

MOV TCON, #50H

(g) Calculate Period

When Timer 0 interrupt occurs, the ISR at 000BH location will be executed. The ISR should read the value from TH1 TL1 and determine the period through the look-up table.

9.12.3 Pulse Width Measurement

For pulse width measurement, the on-chip logic control can be effectively used. As shown in Figures 9.22 to 9.25 and as explained in Section 9.10.1 earlier, the logic to initiate the timer/counter operation may be switched on in many ways. One of the ways is

TR0 = 1 (high)

$\overline{INT0}$ = 1 (high) and Gate = 1 (high)

This may be used to measure the pulse width of any signal in the following manner.

- (a) Connect the signal whose pulse width is to be measured at $\overline{\text{INT0}}$. Program TMOD and TCON to make TR0 and Gate high. Either mode 0 or 1 may be selected.
- (b) On 0 to 1 transition at $\overline{\text{INT0}}$, the timer/counter will start counting at 1/12th of the oscillator frequency.
- (c) Enable the external interrupt at $\overline{\text{INT0}}$.
- (d) On 1 to 0 transition of signal at $\overline{\text{INT0}}$, the timer/counter will stop. It will also cause interrupt at $\overline{\text{INT0}}$.
- (e) The ISR at 0003H must read the timer/counter register TH0 TL0. The operation sequence will be

(i) Connect signal at $\overline{\text{INT0}}$.

(ii) Program TMOD, TCON and IE SFRs

TMOD (Timer/Counter 0 – Gate = 1, $C/T = 0$ Mode = 01) = 0 0 0 0
1 0 0 1 = 09H

TCON (TR0 = 1) = 0 0 0 1 0 0 0 0 = 10H

IE = 1 0 0 0 0 0 0 1 = 81H

MOV TMOD, #09H

MOV TCON, #10H

MOV IE, 81H

(iii) The ISR at 0003H must read the timer registers TH0 and TL0.

If the clock frequency is 12 MHz, the value in TH0 and TL0 represents the pulse width in microsecond.

The Timer/Counter 1 may also be used, instead of Timer/Counter 0. In this case, TR1, $\overline{\text{INT1}}$ and Gate signal corresponding to Timer/Counter 1 will be used.

9.13 POWER DOWN OPERATION

The 8051 provides the facility to save vital data in the Internal RAM, on power failure. The user can provide the backup supply voltage at the RST/VPD pin. In normal operations, the Internal RAM draws its power from V_{CC} . However, if voltage at V_{CC} becomes less than that at RST/VPD, the later becomes the source of power for RAM (Figure 9.33).

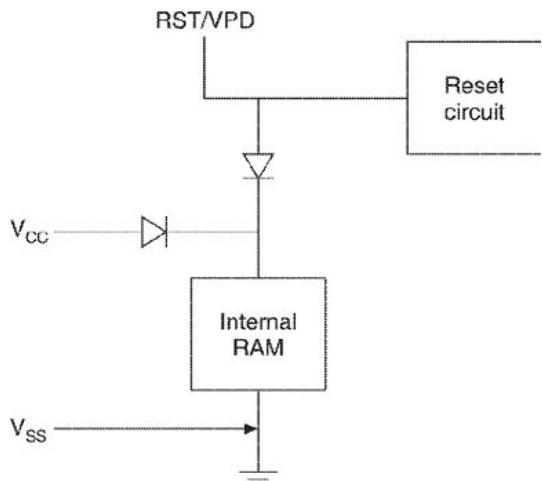


Figure 9.33 Power down operation.

Additionally, a user can design the system to cause the interrupt at $\overline{INT0}$ or $\overline{INT1}$ on power failure. The ISR will save the relevant data in RAM before V_{CC} falls below its operating limit and VPD takes over.

9.14 CONCLUSION

The hardware architecture of the 8051 chip is completely different from that of the microprocessor and other microcomputers like Intel 8048. The effort was to have all possible facilities for embedded control applications on a single chip. The software instructions and case studies in later chapters will further illustrate these.

EXERCISES

1. Will the architecture of the 8051 change if it had 256 byte internal data RAM? Is it possible in the present architecture?
2. An 8051-based system requires 20 KB of external program memory and 15 KB of external data memory. Design the memory interfacing circuit for the system keeping both the memory spaces separately.
3. Design a memory interfacing circuit for the 30 KB combined memory space for external program and data memory. Use a memory chip combination of your choice.
4. Design the interface to the 8051 for the sample-and-hold circuit and the 8-channel analog-to-digital converter. How many port pins will be used and why? You may take any ADC and sample-and-hold chips of your choice.

5. Design the interface for an 8-bit DAC to the 8051. Take any popular DAC chip of your choice.
6. A signal source produces rectangular pulses. The frequency may be between 20 kHz and 25 kHz. The clock frequency of the 8051 is 12 MHz.
 - (a) For frequency measurement, calculate the sample time for 8-bit resolution and values to be loaded in Timer 0 and Timer 1 registers. Is the software timer necessary? Draw a flowchart.
 - (b) For period measurement, calculate the number of periods over which the elapsed time should be measured, i.e. full-scale range of elapsed time, maximum and minimum elapsed time for 8-bit resolution.

FURTHER READING

Designing with the Intel 80C51BH, Application Note AP-252, Intel Corporation, Santa Clara.

Embedded Microcontrollers and Processors, Vols. 1 and 2, Intel Corporation, Santa Clara.

Microcontroller User's Manual, Intel Corporation, Santa Clara.

Vahid, Frank and Tony Givargis, *Embedded System Design*, John Wiley & Sons, 2002.

10

INTEL 8051 MICROCONTROLLER INSTRUCTION SET AND PROGRAMMING

10.1 INTRODUCTION

In Chapter 9, we dealt with the hardware architecture, the memory and the I/O interfacing of the 8051. We have seen that the 8051 microcontroller is designed for embedded system applications, since it contains internal program memory and internal data memory along with bit addressability. However, the designers of the 8051 have also seen to it that it can be used for other applications too, which would require external I/O and memory.

In the present chapter, we shall deal with programming of the 8051. We begin by describing the programmers model, the operand types and then the addressing modes. The instruction set, in general, would then be described along with the examples.

10.2 PROGRAMMERS MODEL OF INTEL 8051

A clear understanding of the 8051 hardware features is essential for its programming and its interaction with the external world. Various hardware resources available in the 8051 microcontroller have already been dealt with in Chapter 9. However, the following resources would be used in programming of the 8051, and thus these represent the programmers model.

- Memory
- Special Function Registers

- Program Status Word

The user will have to refer to these resources quite often during the exercise of writing software.

10.2.1 Memory

The memory map of the 8051 is shown in Figure 10.1. The 8051 can have separate program and data memory, each of 64 KB. Out of 64 KB of program memory, 4 KB of memory is present on-chip as ROM. In addition, it has 256 bytes of on-chip RAM as data memory which contains

- 128 bytes internal data RAM
- 21 Special Function Registers (SFRs).

The internal data RAM (128 bytes) is divided into:

1. 32 bytes for 4 banks of registers R0–R7. The register bank is selected by using two bits RS0 and RS1 in the program status word. These registers can be used as general purpose registers.
2. 16 bytes of direct addressing bits (total 128 bit addresses). These can be directly referred and used in programs.
3. 80 bytes as general purpose RAM.

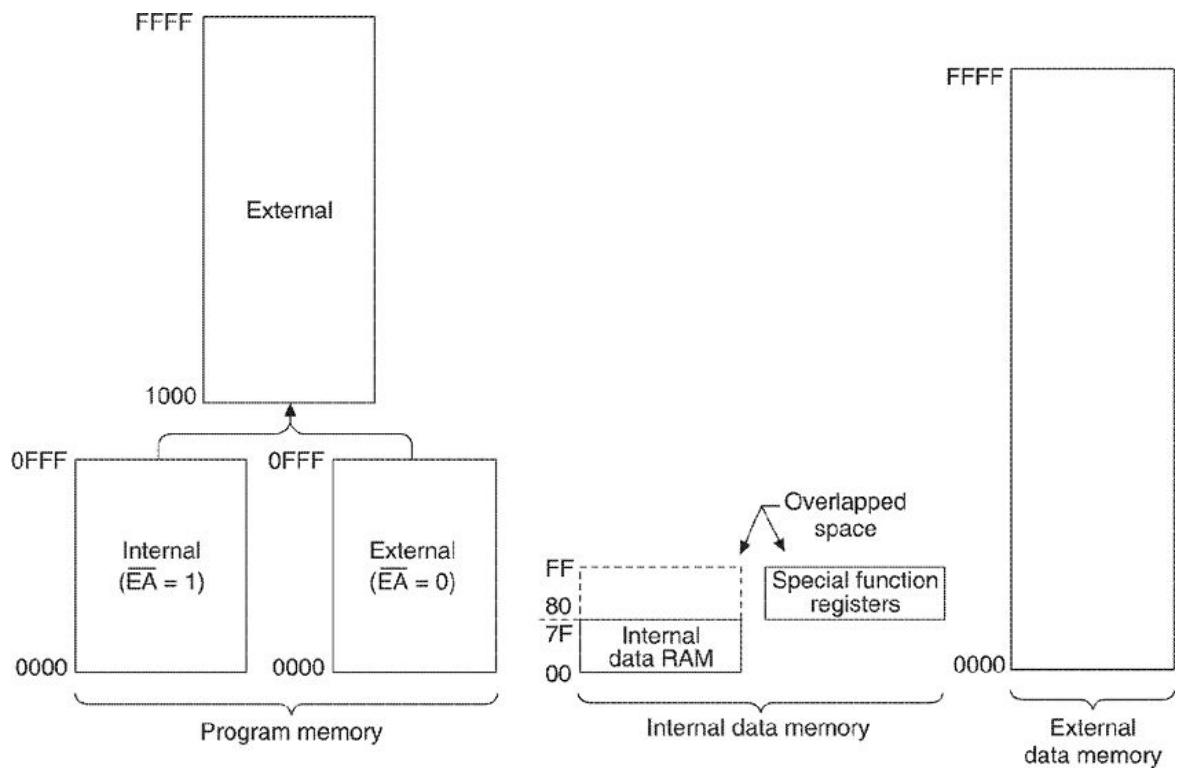


Figure 10.1 Memory mapping of the 8051.

Figure 10.2 shows the details of the 128 bytes internal data RAM.

RAM Byte (MSB)									(LSB)
7FH									127
2FH	7F	7E	7D	7C	7B	7A	79	78	47
2EH	77	76	75	74	73	72	71	70	46
2DH	6F	6E	6D	6C	6B	6A	69	68	45
2CH	67	66	65	64	63	62	61	60	44
2BH	5F	5E	5D	5C	5B	5A	59	58	43
2AH	57	56	55	54	53	52	51	50	42
29H	4F	4E	4D	4C	4B	4A	49	48	41
28H	47	46	45	44	43	42	41	40	40
27H	3F	3E	3D	3C	3B	3A	39	38	39
26H	37	36	35	34	33	32	31	30	38
25H	2F	2E	2D	2C	2B	2A	29	28	37
24H	27	26	25	24	23	22	21	20	36
23H	1F	1E	1D	1C	1B	1A	19	18	35
22H	17	16	15	14	13	12	11	10	34
21H	0F	0E	0D	0C	0B	0A	09	08	33
20H	07	06	05	04	03	02	01	00	32
1FH	Bank 3								31
18H									24
17H	Bank 2								23
10H									16
0FH	Bank 1								15
08H									8
07H	Bank 0								7
00H									0

Figure 10.2 Internal RAM partitioning.

10.2.2 Special Function Registers

The addresses of special function registers (SFRs) are shown in Figure 10.3. Using the SFRs, the different resources of the 8051, like timer/counters, serial ports, interrupts, may be programmed and controlled. Some of the SFRs are bit-addressable as well. These are shown in Figure 10.4.

Symbolic address	Bit address	Byte address
B	255	248
	247	240 (F0H)
ACC	231	224 (E0H)
PSW	215	208 (D0H)
IP	191	184 (B8H)
P3	183	176 (B0H)
IE	175	168 (A8H)
P2	167	160 (A0H)
SBUF		153 (99H)
SCON	159	152 (98H)
P1	151	144 (90H)
TH1		141 (8DH)
TH0		140 (8CH)
TL1		139 (8BH)
TL0		138 (8AH)
TMOD		137 (89H)
TCON	143	136 (88H)
PCON		135 (87H)
DPH		131 (83H)
DPL		130 (82H)
SP		129 (81H)
P0	135	128 (80H)

Special function registers containing direct addressable bits

Figure 10.3 The addresses of special function registers.

Direct byte address	Bit addresses								Hardware register symbol
	(MSB)							(LSB)	
240	F7	F6	F5	F4	F3	F2	F1	F0	B
224	E7	E6	E5	E4	E3	E2	E1	E0	ACC
208	CY	AC	F0	RS1	RS0	OV		P	PSW
	D7	D6	D5	D4	D3	D2	D1	D0	
184				PS	PT1	PX1	PT0	PX0	IP
	—	—	—	BC	BB	BA	B9	B8	
176	B7	B6	B5	B4	B3	B2	B1	B0	P3
	EA			ES	ET1	EX1	ET0	EX0	
168	AF	—	—	AC	AB	AA	A9	A8	IE
160	A7	A6	A5	A4	A3	A2	A1	A0	P2
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI	
152	9F	9E	9D	9C	9B	9A	99	98	SCON
144	97	96	95	94	93	92	91	90	P1
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	
136	8F	8E	8D	8C	8B	8A	89	88	TCON
128	87	86	85	84	83	82	81	80	P0

Figure 10.4 The bit addresses of special function registers.

The special function registers and the resources controlled by them are detailed in Chapter 9 and will be referred to when required.

10.2.3 Program Status Word

The Program Status Word (PSW) contains several status bits that reflect the current state of the CPU. The format of PSW is shown in Figure 10.5. The PSW is one of the Special Function Registers and contains the carry bit, the auxiliary carry bit (for BCD operations), two register select bits, the overflow flag bit, a parity bit, and two user definable status flag bits. Following are the brief descriptions of these bits:

- CY (PSW.7) is set, if the operation results in a carry out of (during addition) or a borrow into (during subtraction) the high-order bit of the result; otherwise CY is cleared.

- AC (PSW.6) is set, if the operation results in a carry out of low-order 4 bits of the result (during addition) or a borrow from the high-order bits into the low-order 4 bits (during subtraction); otherwise AC is cleared.
 - RS1, RS0 (PSW.4, PSW.3) represent the current register bank in the Internal Data RAM selected.

RS1	RS0	
0	0	Register bank 0
0	1	Register bank 1
1	0	Register bank 2
1	1	Register bank 3

- OV (PSW.2) is set, if the operation results in a carry into the high-order bit of the result but not a carry out of the high-order bit, or vice-versa; otherwise OV is cleared. OV has a significant role in two's-complement arithmetic, since it becomes set when the signed result cannot be represented in 8 bits.
 - P (PSW.0) is set, if the modulo 2 sum of the eight bits in the accumulator is 1 (odd parity); otherwise P is cleared (even parity). When a value is written to the PSW register, the P bit remains unchanged as it always reflects the parity of the accumulator contents.

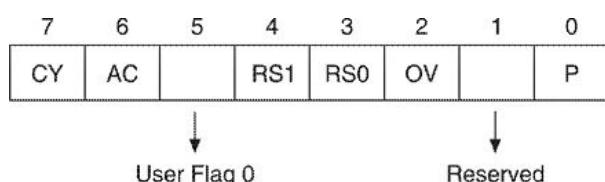


Figure 10.5 The format of the Program Status Word.

10.3 OPERAND TYPES

The 8051 architecture supports the following data types:

- Bytes
 - Short integers
 - Bits

Bytes and short integers are 8-bit variables and can be placed anywhere in memory, without any alignment restrictions.

The 8051 provides extensive bit operation facility. The bit processor (though a part of the 8051 architecture), may be considered an independent processor since it has its own instruction set and its own independent resources like accumulator (the carry flag), the bit addressable RAM and I/O. 128 bits within the Internal Data RAM and 128 bits within the SFRs may be addressed directly.

The 8051 instructions do not facilitate 16-bit operations and, therefore, 16-bit operand addressing is not possible in these instructions. Only the MUL instruction produces the 16-bit result stored in register B (higher byte) and register A (lower byte). Thus, it is the programmer's task to represent the 16-bit data and carry out arithmetic/logic operations as in the case of the 8085. The 16-bit address can, however, be loaded in Data pointer (DPTR) through MOV instructions.

10.4 OPERAND ADDRESSING

There are five ways of addressing operands.

1. Register addressing
2. Direct addressing
3. Register-indirect addressing
4. Immediate addressing
5. Base register plus index register indirect addressing

10.4.1 Register Addressing

Register addressing permits access to eight registers (R0–R7) of the register bank. There are four banks of eight registers. One of the four banks is selected by a 2-bit field in PSW (Program Status Word). Other registers used are A, B, AB and DPTR. For example,

MOV A, Rn

Move contents of register Rn to accumulator.

Hence, after the execution of this instruction, the registers Rn and A have the original contents of register Rn (Figure 10.6).

10.4.2 Direct Addressing

In direct addressing, the address of the operand is specified in the instruction. Direct addressing has operands as byte or bit. Direct

addressing of byte provides operation on one of the following.

1. Lower 128 bytes of internal data RAM
2. Special function registers

Direct bit addressing provides operation on the following.

1. 128 bits subset of internal data RAM (20H to 2FH)
2. 128 bits subset of special function register address space (80H to FFH)

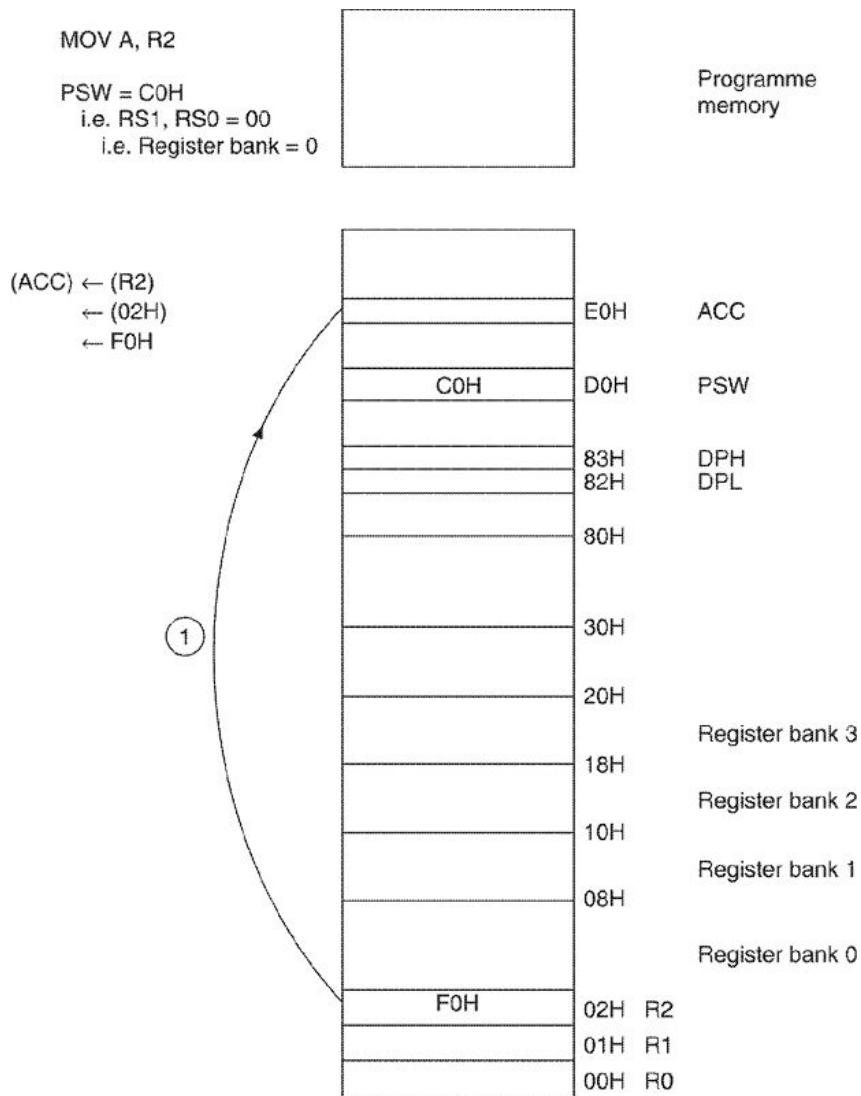


Figure 10.6 An example of register addressing.

For example, consider the instruction

`MOV A, Direct`

In the above instruction, the byte variable indicated by Direct (which is an address location) is copied into register A (Figure 10.7).

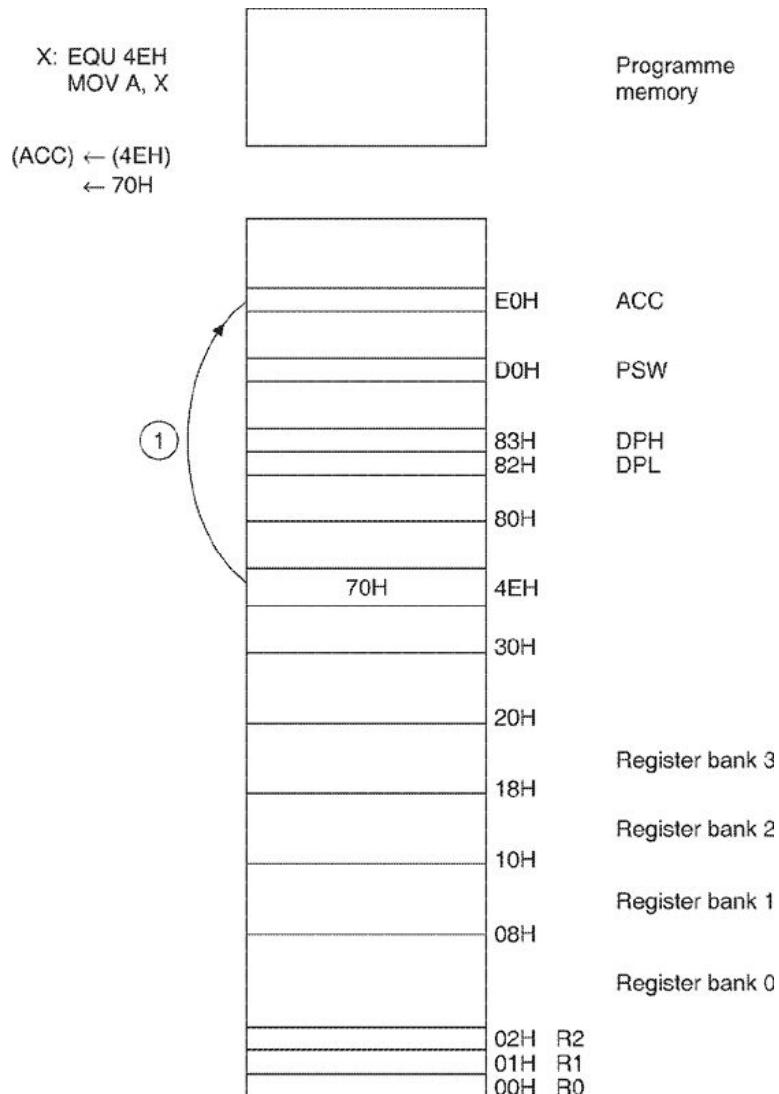


Figure 10.7 An example of direct addressing.

10.4.3 Indirect Addressing

In this addressing, the address of the operand is not specified directly. Instead, the address of the operand is specified as the contents of the register mentioned in the instruction. For example, in the instruction `MOV A, @Ri`, the contents of register Ri give the address of the location from where the operand is picked up. The above instruction causes the operand to be picked up from the mentioned location for transfer to register A. The operation sequence has been explained in Figure 10.8 for the instruction `MOV A, @R0`.

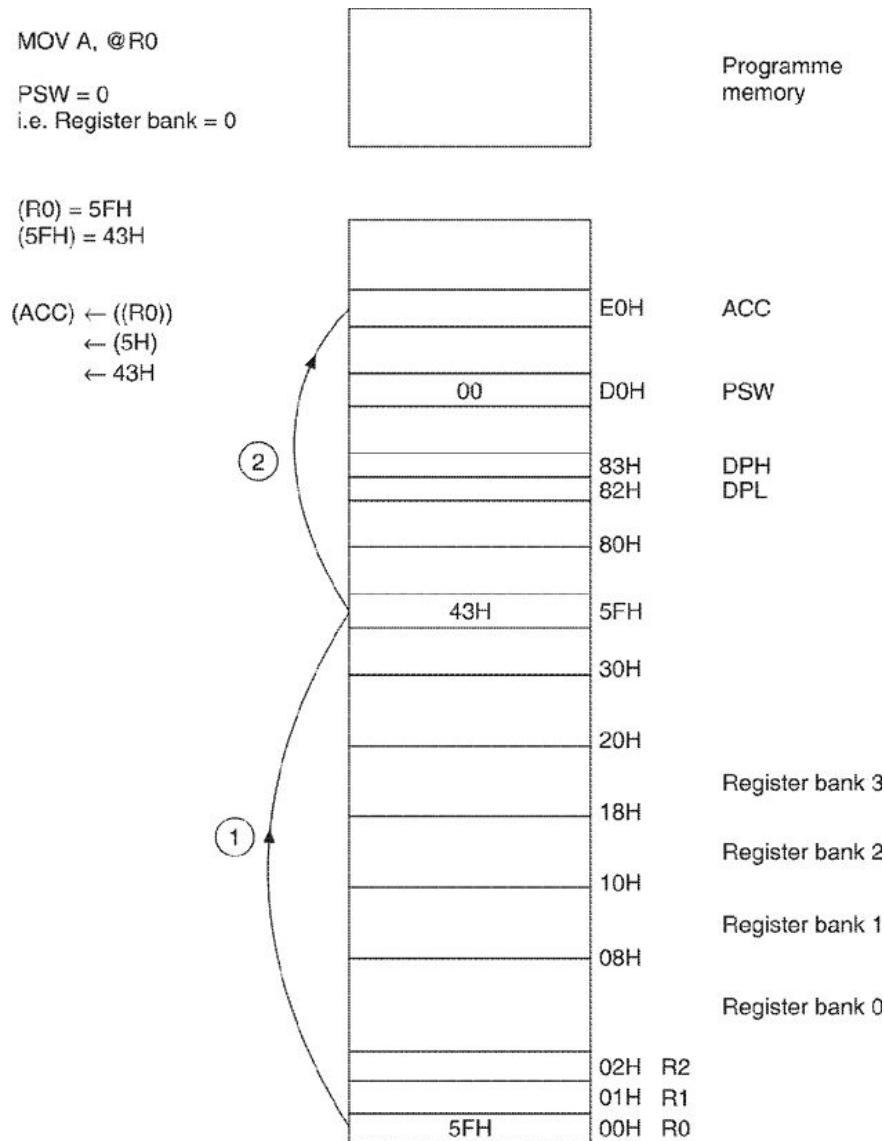


Figure 10.8 An example of indirect addressing.

This type of instruction is useful when the same set of operations is performed on different data sets stored in memory. In the present case either R0 or R1 may be used for accessing the locations within the 256 byte locations.

However, register indirect addressing is also used for accessing external data memory. The 16-bit pointer (DTPR) can be used for accessing any location within the full 64 KB memory space.

10.4.4 Immediate Addressing

In this case the operand on which the operation has to be performed according to the instruction, is specified in the instruction itself. For example, the instruction ADD A, #data, adds the 8-bit data specified in

the instruction to the contents of register A and the result is placed in register A itself (Figure 10.9).

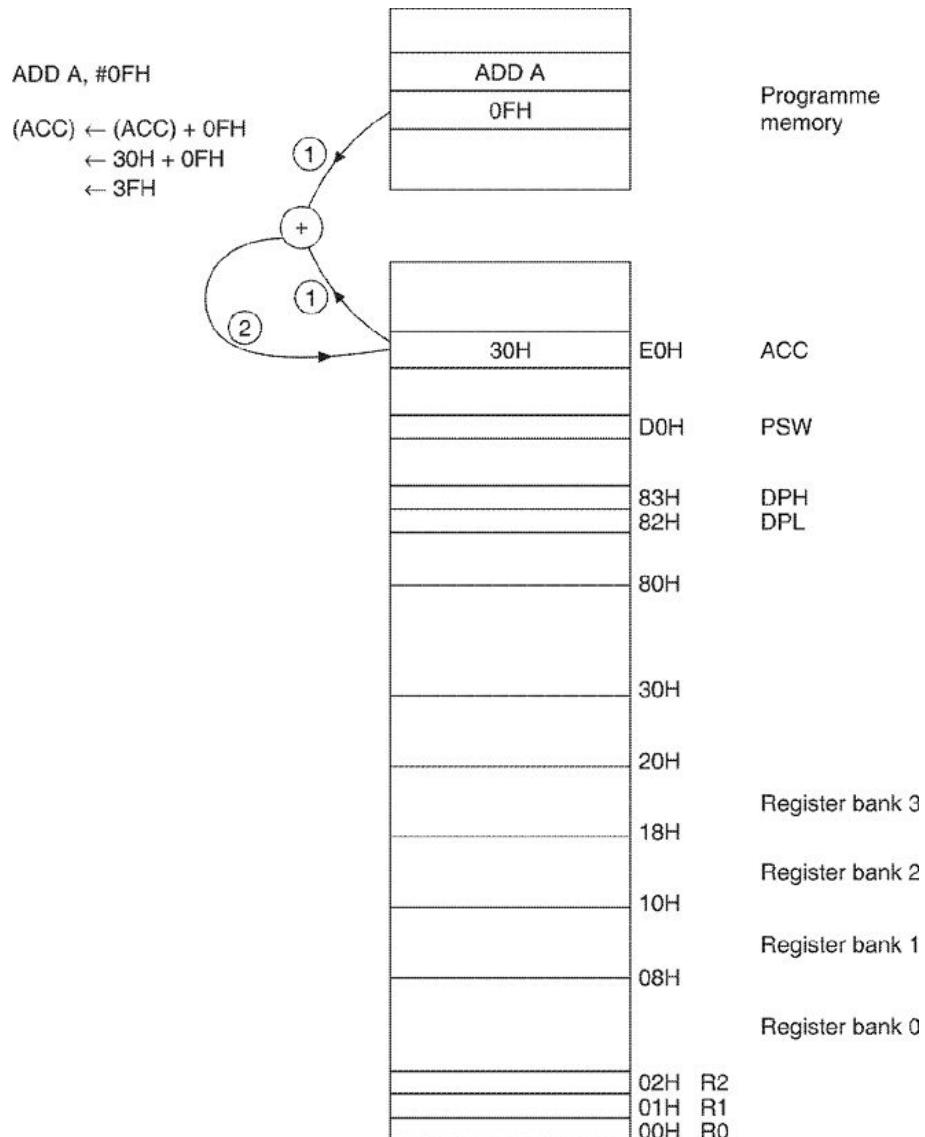


Figure 10.9 An example of immediate addressing.

10.4.5 Base Register Plus Index Register Indirect Addressing

This is an indirect instruction used to access the program memory. In the instruction the operand on which the operation is to be performed, is not specified directly. The summation of contents of base register and index register determines the operand address. The DPTR or PC register may act as the base register and register A acts as the index register. Thus the base register (DPTR or PC) and the index register A are added to get the memory location. The operand is the contents of the memory location calculated above. For example, the instruction MOVC A, @ A+DPTR

moves a byte from a specified address to register A. The address of the byte to be moved in register A is the sum of the original 8 bits of accumulator contents and 16 bits of base register (which is the DPTR).

The operation sequence for instruction MOVC A, @A+DPTR is explained in Figure 10.10.

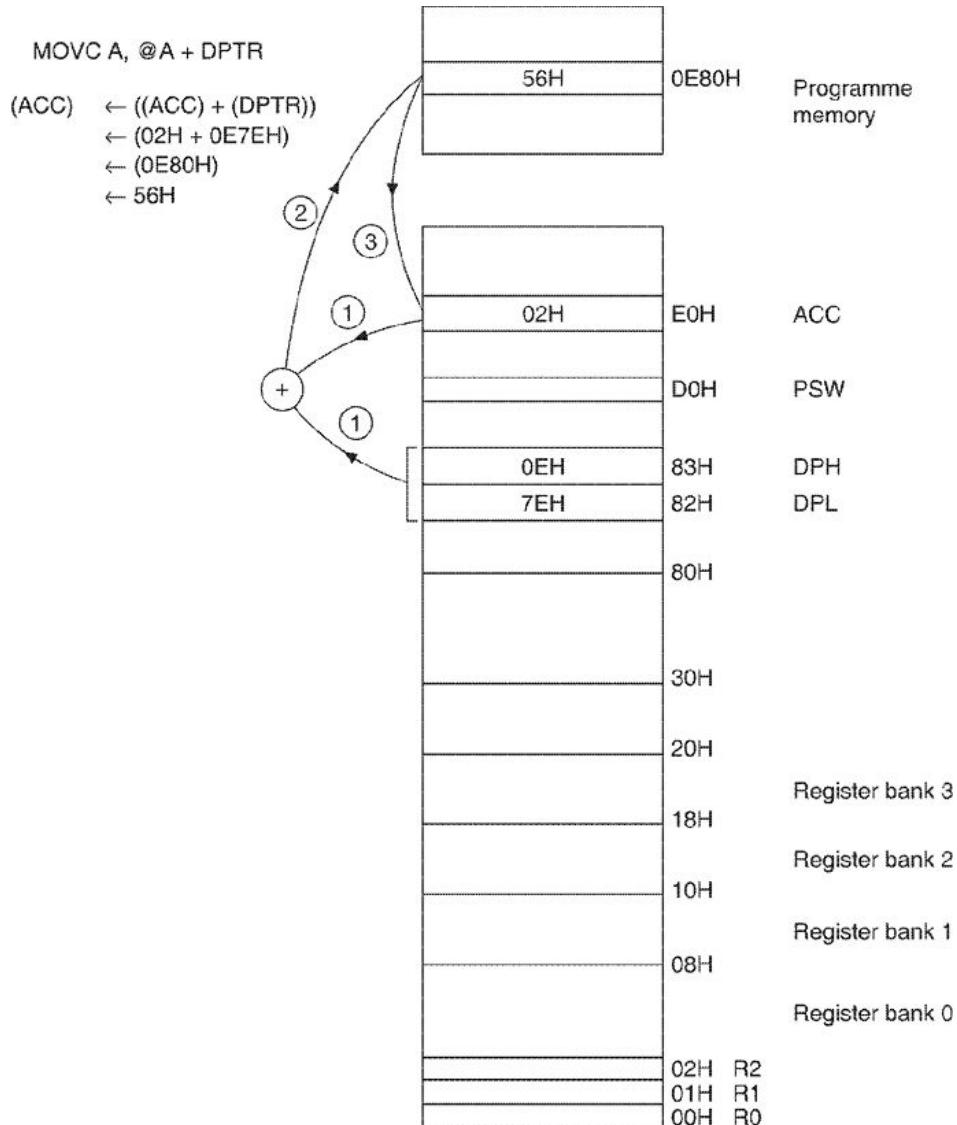


Figure 10.10 An example of base register plus index register indirect addressing.

10.5 DATA TRANSFER INSTRUCTIONS

Data transfer operations are divided into three classes.

- General-purpose transfers
- Accumulator-specific transfers
- Address-object transfers

None of the above operations affect the flag settings except a POP or MOV into the PSW.

10.5.1 General-purpose Transfers

Three general-purpose data transfer operations are provided, which may be applied to most operands; though there are some specific exceptions.

- MOV performs a bit or a byte transfer from the source operand to the destination operand.
- PUSH increments the SP register and then transfers a byte from the source operand to the stack element, currently addressed by SP.
- POP transfers a byte operand from the stack element addressed by the SP register to the destination operand and then decrements SP.

PUSH and POP operation are explained in Chapter 9 under Section 9.4.1.

MOV <dest.byte>, <scr.byte> (Move byte variable)

The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected. This is by far the most flexible operation. Fifteen combinations of the source and destination addressing modes are allowed.

MOV A, Rn

(A) □ (Rn)

MOV A, direct

(A) □ (direct)

MOV A, @Ri (where Ri = R0 or R1)

(A) □ ((Ri))

MOV A, #data

(A) □ #data

MOV Rn, A

(Rn) □ (A)

MOV Rn, direct

(Rn) □ (direct)

MOV Rn, #data

(Rn) \square #data

MOV direct, A

(direct) \square (A)

MOV direct, Rn

(direct) \square (Rn)

MOV direct, direct

(direct) \square (direct)

MOV direct, @Ri (where Ri = R0 or R1)

(direct) \square ((Ri))

MOV direct, #data

(direct) \square #data

MOV @Ri, A (where Ri = R0 or R1)

((Ri)) \square (A)

MOV @Ri, direct (where Ri = R0 or R1)

((Ri)) \square (direct)

MOV @Ri, #data (where Ri = R0 or R1)

((Ri)) \square #data

It may be noted that the contents of a register in the register bank cannot be transferred to another register in the register bank, since the instruction MOV Rm, Rn is not present.

It must also be noted that if any data is moved to byte location 208 (i.e. PSW) then the flag settings may change; otherwise, these instructions do not affect the flags.

MOV <dest-bit>, <src-bit> (Move bit data)

The Boolean variable, indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

MOV C, bit

(C) \square (bit)

MOV bit, C

(bit) \square (C)

PUSH direct (Push onto stack)

The stack pointer is incremented by one. The contents of the indicated variable are then copied into the internal RAM location addressed by the stack pointer. The instruction may be used to modify the contents of the byte location 208 (i.e. PSW) in case the stack locations coincide with the SFR locations; otherwise no flags are affected.

$$\begin{aligned} & (\text{SP}) \square (\text{SP}) + 1 \\ & ((\text{SP})) \square (\text{direct}) \end{aligned}$$

POP direct (Pop from stack)

The contents of the internal RAM location addressed by the stack pointer are read, and the stack pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. The instruction may be used to modify the contents of byte location 208 (i.e. PSW), in case the addressed memory location coincides with the PSW register location; otherwise no flags are affected.

$$\begin{aligned} & (\text{direct}) \square ((\text{SP})) \\ & (\text{SP}) \square (\text{SP}) - 1 \end{aligned}$$

10.5.2 Accumulator-specific Transfers

Four accumulator-specific transfer operations are provided.

- XCH exchanges the byte source operand with register A (accumulator).
- XCHD exchanges the low-order nibble of the byte source operand with the low-order nibble of register A.
- MOVX performs a byte move between the External Data Memory and register A. The external address can be specified by the DPTR register (16 bits) or the R1 or R0 register (8 bits).
- MOVC performs the move of a byte from the program memory to register A.

XCH A, <byte> (Exchange accumulator with byte variable)

The XCH exchanges the contents of ACC with those of the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

$$\begin{aligned} & \text{XCH A, Rn} \\ & (\text{A}) \square (\text{Rn}) \end{aligned}$$

XCH A, direct

(A) \square (direct)

XCH A, @Ri (where Ri = R0 or R1)

(A) \square ((Ri))

The instruction may be used to modify the contents of the byte location 208, i.e. PSW, in case the addressed memory location coincides with the PSW register location; otherwise no flags are affected.

XCHD A, @Ri (Exchange digit) (where Ri = R0 or R1)

The XCHD exchanges the low-order nibble of the accumulator (bits 3–0) with that of the internal RAM location indirectly addressed by the specified register. The instruction may be used to modify the contents of the byte location 208, i.e. PSW, in case the addressed memory location coincides with the PSW register location; otherwise no flags are affected.

(A)3–0 \square ((Ri))3–0

MOVX <dest-byte>, <src-byte> (Move external)

The MOVX instructions transfer data between the accumulator and a byte of the external data memory. There are two types of instructions, differing in whether they provide an 8-bit or a 16-bit indirect address to the external data RAM. In the first type, the contents of Ri (i.e. R0 or R1) in the current register bank provide an 8-bit address for the external I/O expansion decoding or a relatively small RAM array. In the second type of MOVX instruction, the data pointer generates a 16-bit address.

MOVX A, @Ri (where Ri = R0 or R1)

(A) \square ((Ri))

MOVX @Ri, A (where Ri = R0 or R1)

((Ri)) \square (A)

MOVX A, @DPTR

(A) \square ((DPTR))

MOVX @DPTR, A

(DPTR) \square (A)

MOVC A, @A + <base-reg> (Move code byte)

The MOVC instructions load the accumulator with a code byte, or a constant from the program memory. No flags are affected.

MOVC A, @A + DPTR

(A) \square ((A) + (DPTR))

$\text{MOVC A, } @A + \text{PC}$
 $(\text{PC}) \square (\text{PC}) + 1$
 $(A) \square ((A) + (\text{PC}))$

10.5.3 Address-object Transfer

MOV DPTR, #data loads 16 bits of immediate data into a pair of destination registers, DPH and DPL.

MOV DPTR, #data16 (Load data pointer with a 16-bit constant)

The data pointer is loaded with the 16-bit constant indicated. No flags are affected.

$(\text{DPTR}) \square \#data16$

Some examples are now discussed below:

EXAMPLE 10.1

The internal RAM location 30H holds 40H. The value stored in RAM location 40H is 10H. What will be the value in locations 30H and 40H after the following instructions are executed?

```

MOV R0, #30H
MOV A, @R0
MOV R1, A
MOV A, #7FH
MOV @R1, A
XCHD A, @R0

```

Solution:

We shall take each instruction in sequence to understand the operation.

MOV R0, #30H	; (R0) = 30H
MOV A, @R0	; (A) = 40H
MOV R1, A	; (R1) = 40H
MOV A, #7FH	; (A) = 7FH
MOV @R1, A	; RAM (40H) = 7FH
XCHD A, @R0	; RAM (30H) = 4FH, (A) = 70H

EXAMPLE 10.2

The carry flag is originally set. The data present at the input port 3 is C5H. The data previously written to output port 1 is 35H. Calculate the port 1 contents after the execution of the following instructions.

MOV P1.3, C

MOV C, P3.3

MOV P1.2, C

Solution:

Let us work out as follows:

Originally (P1) = 0011 0101B, (P3) = 1100
0101B, (C) = 1

After the first instruction (P1) = 0011 1101B, (P3) = 1100
0101B, (C) = 1

After the second instruction (P1) = 0011 1101B, (P3) = 1100
0101B, (C) = 0

After the third instruction (P1) = 0011 1001B, (P3) = 1100
0101B, (C) = 0

Thus, port 1 will become 39H.

EXERCISES

1. R0 contains 30H. The accumulator holds the value 3FH. The internal RAM locations 30H and 3FH hold the values 75H and 95H respectively. What will be the contents of ACC and location 30H after the execution of the following instructions?

XCH A, @R0

MOV A, @R0

2. The program memory locations 1025, 1026, 1027 have the constants 0FH, 0EH and 00 stored respectively. The stack pointer originally contains the value 32H, and the internal RAM locations 30H through 32H contain the values 20H, 23H, and 10H respectively. What will be the values stored in DPL, DPH and ACC after the execution of the following instructions?

MOV A, #03H

POP DPH

POP DPL

MOVC A, @A + DPTR

10.6 ARITHMETIC INSTRUCTIONS

The 8051 provides the basic mathematical operations like addition, subtraction, multiplication, division, increment, decrement, etc. Only the 8-bit operations using unsigned arithmetic are supported directly. The

overflow flag permits the addition and subtraction operations to serve both unsigned and signed binary integers. A correction operation is also provided to allow the arithmetic to be performed directly on the packed decimal (BCD) representations.

Note: Some instructions like INC (increment), DEC (decrement) may also be used to modify the output port. In such cases the value used as the original port data will be read from the output data latch, not from the input pins.

10.6.1 Addition

There are four addition operations.

- ADD performs an addition between the register A and the second source operand.
- ADDC (add with carry) performs an addition between the register A and the second source operand; adds 1 if the C flag is found previously set.
- DA (decimal-add-adjust for BCD addition) performs a correction to the sum which resulted from the binary addition of two two-digit decimal operands. The packed decimal sum formed by DA is returned to A. The carry flag is set if the BCD result is greater than 99; or else it is cleared.
- INC (increment) performs an addition of the source operand and 1.

ADD A, <src.byte> (Add)

ADD adds the byte variables indicated to the accumulator, leaving the result in the accumulator. Four source operand addressing modes are allowed—register, direct, register indirect, and immediate.

ADD A, Rn

$$(A) \square (A) + (Rn)$$

ADD A, direct

$$(A) \square (A) + (\text{direct})$$

ADD A, @Ri (where Ri = R0 or R1)

$$(A) \square (A) + ((Ri))$$

ADD A, #data

$$(A) \square (A) + \#data$$

ADDC A, <src.byte> (Add with carry)

The ADDC simultaneously adds the byte variables indicated, the carry flag and the accumulator contents, leaving the result in the accumulator. Four source operand addressing modes are allowed—register, direct, register indirect, and immediate.

ADDC A, Rn

$$(A) \square (A) + (C) + (Rn)$$

ADDC A, direct

$$(A) \square (A) + (C) + (\text{direct})$$

ADDC A, @Ri (where Ri = R0 or R1)

$$(A) \square (A) + (C) + ((Ri))$$

ADDC A, #data

$$(A) \square (A) + (C) + \#data$$

DA A (Decimal-adjust accumulator for addition)

The DA A adjusts the 8-bit value in the accumulator, resulting from the earlier addition of two variables (each in packed-BCD format), producing two 4-bit digits. Any ADD or ADDC instruction may have been used to perform the addition. The algorithm followed is as follows:

1. If the value of the lower nibble in ACC is greater than 9, or if AC flag is set, then 6 is added to ACC.
2. If the value of the higher nibble is now greater than 9, or if CY flag is set, then 6 is added to the higher nibble of ACC.

All flags are affected.

If $[(A3-0) > 9] > [(AC) = 1]$

Then $(A3-0) \square (A3-0) + 6$

and

If $[(A7-4) > 9] > [(C) = 1]$

Then $(A7-4) \square (A7-4) + 6$

INC <byte> (Increment)

INC increments the indicated variable by 1. No flags are affected.

Four addressing modes are allowed—accumulator, register, direct, or register indirect.

INC A

$$(A) \square (A) + 1$$

INC Rn

$$(Rn) \square (Rn) + 1$$

INC direct

$$(\text{direct}) \square (\text{direct}) + 1$$

INC @Ri (where Ri = R0 or R1)

$$((Ri)) \square ((Ri)) + 1$$

INC DPTR (Increment Data Pointer)

Increments the 16-bit data pointer by 1. No flags are affected.

$$(\text{DPTR}) \square (\text{DPTR}) + 1$$

10.6.2 Subtraction

There are two subtraction operations.

- SUBB (subtract with borrow) performs a subtraction of the second source operand from the first operand (the accumulator), subtracts 1 if the C flag is found previously set.
- DEC (decrement) performs a subtraction of 1 from the source operand.

SUBB A, <src-byte> (Subtract with borrow)

SUBB subtracts the indicated variable and the carry flag together from the accumulator, leaving the result in the accumulator. The source operand allows four addressing modes—register, direct, register indirect, and immediate.

SUBB A, Rn

$$(A) \square (A) - (C) - (Rn)$$

SUBB A, direct

$$(A) \square (A) - (C) - (\text{direct})$$

SUBB A, @Ri (where Ri = R0 or R1)

$$(A) \square (A) - (C) - ((Ri))$$

SUBB A, #data

$$(A) \square (A) - (C) - \#data$$

DEC byte (Decrement)

The variable indicated is decremented by 1. No flags are affected. Four operand addressing modes are allowed—accumulator, register, direct, and register indirect.

DEC A

$$(A) \square (A) - 1$$

DEC R_n

$$(R_n) \square (R_n) - 1$$

DEC direct

$$(\text{direct}) \square (\text{direct}) - 1$$

DEC @R_i (where R_i = R₀ or R₁)

$$((R_i)) \square ((R_i)) - 1$$

10.6.3 Multiplication

MUL performs an unsigned multiplication of register A by register B, returning a double-byte result.

MUL AB (Multiply)

The MUL AB multiplies the unsigned 8-bit integers in the accumulator and register B. The low-order byte of the 16-bit product is left in the accumulator, and the high-order byte in register B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

$$\begin{aligned} (A)_{7-0} &\square (A) \square (B) \\ (B)_{15-8} & \end{aligned}$$

10.6.4 Division

The DIV performs an unsigned division of register A by register B.

DIV AB (Divide)

The DIV AB divides the unsigned 8-bit integer in the accumulator by the unsigned 8-bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and the overflow flags will be cleared. An exception is: if B had originally contained 00H, the values returned to the accumulator and B register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

$$\begin{aligned} (A)_{15-8} \\ \square (A)/(B) \end{aligned}$$

(B) 7–0

Some examples are given below for better understanding.

EXAMPLE 10.3

Register R0 contains 7EH (01111110B). The internal RAM locations 7EH and 7FH contain FFH and 40H respectively. The following instruction sequence is executed

```
INC      @R0  
MOV A,  @R0  
INC      R0  
INC      @R0
```

What will be the contents of RAM locations 7EH, 7FH, register R0 and register A?

Solution:

Let us work out as follows:

Originally (R0) = 7EH, (A) = ?, RAM(7EH) = FFH,
RAM(7FH) = 40H

After the first instruction (R0) = 7EH, (A) = ?, RAM(7EH) = 00H,
RAM(7FH) = 40H

After the second instruction (R0) = 7EH, (A) = 00H, RAM(7EH) = 00H,
RAM(7FH) = 40H

After the third instruction (R0) = 7FH, (A) = 00H, RAM(7EH) =
00H, RAM(7FH) = 40H

After the fourth instruction (R0) = 7FH, (A) = 00H, RAM(7EH) =
00H, RAM(7FH) = 41H

EXAMPLE 10.4

The register R2 holds 34H and the carry flag is set. What will be the value in the register A and the status of flags C, AC and OV, after the execution of the following instructions?

```
MOV A, #5FH  
SUBB A, R2
```

Solution:

Originally (A) = ?, (R2) = 34H, (C) = 1, (AC) =
?, (OV) = ?

After the first instruction (A) = 5FH, (R2) = 34H, (C) = 1, (AC)
= ?, (OV) = ?

After the second instruction (A) = 2AH, (R2) = 34H, (C) = 0, (AC) = 0, (OV) = 0

EXERCISES

1. Register B holds the value 18H. What will be values in registers A and B after the following instructions are executed?

MOV A, #30H
MUL AB
ADD A, #18H
DIV AB

2. The accumulator holds the value 76H, representing the packed BCD digits of the decimal number 76. Register 3 contains the value 43H, representing the packed BCD digits of the decimal number 43. The carry flag is set. What will be value in ACC after the following instructions are executed?

INC A
ADDC A, R3
DA A

10.7 LOGIC INSTRUCTIONS

The 8051 performs the basic logic operations on both bit and byte operands.

10.7.1 Single-operand Operations

There are seven single-operand logical operations as follows:

- CLR is used to set either the register A, the carry, or any direct addressed bit to 0.
- SETB sets either the carry or any direct addressed bit to 1.
- CPL either forms the 1's complement of the operand in register A or the 1's complement of the carry or any direct addressed bit.
- RL, RLC, RR, RRC, SWAP. Five rotate operations can be performed on register A—RL (rotate left), RR (rotate right), RLC (rotate left through carry), RRC (rotate right through carry), and SWAP (swap nibbles). For RLC and RRC, the C flag becomes equal to the last bit rotated out. SWAP rotates the

register A four places left to exchange bits 3 through 0 with bits 7 through 4.

Note: The CPL bit and CLR bit instructions may be used to modify an output pin. In such cases, the value used as the original data will be read from the output data latch, not the input pin.

CLR A (Clear accumulator)

The accumulator is cleared (all bits set to 0). No flags are affected.

(A) \square 0

CLR bit (Clear bit)

The indicated bit is cleared (reset to 0). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.

CLR C

(C) \square 0

CLR bit

(bit) \square 0

SETB <bit> (Set bit)

The SETB sets the indicated bit to 1. The SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

SETB C

(C) \square 1

SETB bit

(bit) \square 1

CPL A (Complement accumulator)

Each bit of the accumulator is logically complemented (1's complement). No flags are affected.

(A) \leftarrow (\bar{A})

CPL bit (Complement bit)

The bit variable specified is complemented. A bit which had been 1 is changed to 0 and vice-versa. No other flags are affected. The CPL can operate on the carry or any directly addressable bit.

CPL C

(C) \leftarrow (\bar{C})

CPL bit

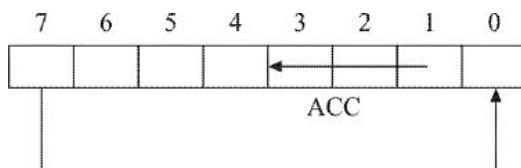
$(\text{bit}) \leftarrow (\overline{\text{bit}})$

RL A (Rotate accumulator left)

The eight bits in the accumulator are rotated 1 bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

$$(A_n + 1) \square (A_n) \quad n = 0-6$$

$$(A_0) \square (A_7)$$



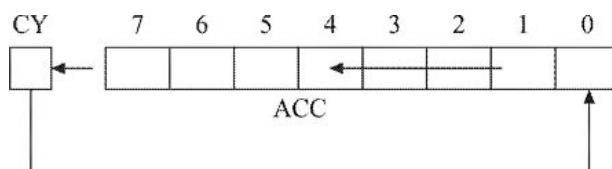
RLC A (Rotate accumulator left through the carry flag)

The eight bits in the accumulator and the carry flag are together rotated 1 bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

$$(A_n + 1) \square (A_n) \quad n = 0-6$$

$$(A_0) \square (C)$$

$$(C) \square (A_7)$$

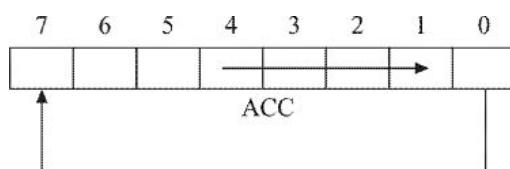


RR A (Rotate accumulator right)

The eight bits in the accumulator are rotated 1 bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

$$(A_n) \square (A_{n+1}) \quad n = 0-6$$

$$(A_7) \square (A_0)$$



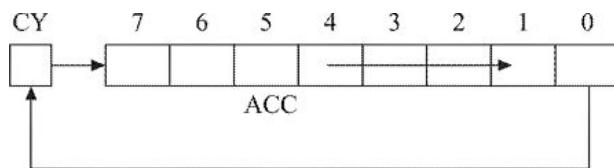
RRC A (Rotate accumulator right through carry flag)

The eight bits in the accumulator and the carry flag are together rotated 1 bit to the right. Bit 0 moves into the carry flag, the original value of the carry flag moves into the bit 7 position. No other flags are affected.

$$(A_n) \square (A_{n+1}) \quad n = 0-6$$

$$(A_7) \square (C)$$

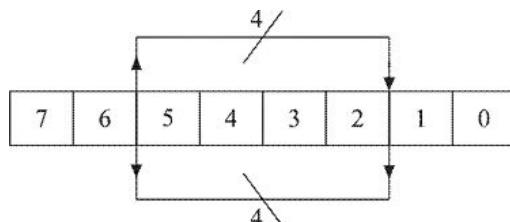
(C) \square (A0)



SWAP A (Swap nibbles within the accumulator)

SWAP A interchanges the low- and high-order nibbles (4-bit fields) of the accumulator (bits 3–0 and bits 7–4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected.

(A3–0) \square (A7–4)



10.7.2 Two-operand Operations

Three two-operand logical instructions ANL, ORL and XRL are provided to perform AND, OR and XOR operations respectively on bit as well as byte operands.

Note: These instructions may be used to modify an output port. In such cases, the value used as the original port data will be read from the output data latch, not the input pins.

ANL <dest.byte>, <src.byte> (Logical AND for byte variables)

ANL performs the bit-wise logical AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or the immediate data.

ANL A, Rn

(A) \square (A) \square (Rn)

ANL A, direct

(A) \square (A) \square (direct)

ANL A, @Ri (where Ri = R0 or R1)

(A) \square (A) \square ((Ri))

ANL A, #data
 (A) \square (A) \square #data
 ANL direct, A
 (direct) \square (direct) \square (A)
 ANL direct, #data
 (direct) \square (direct) \square #data

ANL C, <src.bit> (Logical AND for bit variables)

The carry flag is modified by ANDing with the source bit or its logical complement. No other flags are affected. Only the direct bit addressing is allowed for the source operand.

ANL C, bit
 (B) \square (C) \square (bit)
 ANL C, /bit
 $(B) \leftarrow (C) \wedge (\overline{\text{bit}})$

ORL <dest-byte>, <src-byte> (Logical OR for byte variables)

The ORL performs the bit-wise logical OR operation between the indicated variables, storing the results in the destination byte. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

ORL A, Rn
 (A) \square (A) $>$ (Rn)
 ORL A, direct
 (A) \square (A) $>$ (direct)
 ORL A, @R_i (where R_i = R0 or R1)
 (A) \square (A) $>$ ((R_i))
 ORL A, #data
 (A) \square (A) $>$ #data
 ORL direct, A
 (direct) \square (direct) $>$ (A)
 ORL direct, #data
 (direct) \square (direct) $>$ #data

ORL C, <src-bit> (Logical OR for bit variables)

The carry flag is modified by ORing with the source bit or its logical complement. No other flags are affected.

ORL C, bit

(C) \square (C) > (bit)

ORL C, /bit

(C) \square (C) > $(\overline{\text{bit}})$

XRL <dest-byte>, <scr-byte> (Logical XOR for byte variables)

The XRL performs the bit-wise logical XOR operation between the indicated variables, storing the results in the destination. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register indirect, or immediate addressing. When the destination is a direct address, the source can be the accumulator or immediate data.

XRL A, Rn

(A) \square (A) \vee (Rn)

XRL A, direct

(A) \square (A) \vee (direct)

XRL A, @Ri (where Ri = R0 or R1)

(A) \square (A) \vee ((Ri))

XRL A, #data

(A) \square (A) \vee #data

XRL direct, A

(direct) \square (direct) \vee (A)

XRL direct, #data

(direct) \square (direct) \vee #data

Some examples are discussed below.

EXAMPLE 10.5

What will be the contents of the accumulator after the execution of the following instructions?

MOV A, #CDH
RR A
CPL A
SWAP A

Solution:

Originally (A) = ?

After the first instruction	(A) = CDH (1100 1101B)
After the second instruction	(A) = E6H (1110 0110B)
After the third instruction	(A) = 19H (0001 1001B)
After the fourth instruction	(A) = 91H (1001 0001B)

EXAMPLE 10.6

Set the carry flag if and only if, (P2.7) = 1, (ACC.0) = 1, and (OV) = 0.

Solution:

MOV C, P2.7	; Load carry with P2.7 pin state
ANL C, ACC.0	; And carry with ACC.0
ANL C, /OV	; And carry with the inverse of OV flag

EXERCISES

1. If the accumulator holds C3H, register R0 holds AAH, register R3 holds C9H and register R4 holds D7H, what will be the contents of ACC after the following instructions are executed?

ANL A, R0
ORL A, R4
XRL A, R3

2. If port 1 holds the value 69H (0110 1001B), what will be the value at P1 after the execution of following instructions?

MOV A, #C5H
CLR P1.0
CPL P1.4
ANL A, P1
MOV P1, A

10.8 CONTROL TRANSFER INSTRUCTIONS

There are three classes of control transfer operations.

1. Unconditional calls, returns and jumps
2. Conditional jumps
3. Interrupts

All control transfer operations cause, some upon a specific condition, the program execution to continue at a non-sequential location in the program memory.

10.8.1 Unconditional Calls, Returns and Jumps

Unconditional calls, returns and jumps transfer the control from the current value of the Program Counter to the target address. Both direct and indirect transfers are supported. The three transfer operations are described below.

- ACALL and LCALL push the address of the next instruction onto the stack (PCL to low-order address, PCH to high-order address) and then transfer the control to the target address. Absolute Call is a 2-byte instruction and used when the target address is in the current 2K page. Long Call is a 3-byte instruction that addresses the full 64K program space. In ACALL, the immediate data (i.e. an 11-bit address field) is concatenated to the five most significant bits of the PC (which is pointing to the next instruction). If ACALL is in the last 2 bytes of a 2K page, then the call will be made to the next page since the PC will have to be incremented to the next instruction prior to the execution.
- RET (Return from subroutine) and RETI (Return from Interrupt) transfer control to return address saved on the stack and decrement SP register by 2.
- AJMP, LJMP and SJMP are unconditional branch instructions used to transfer control to the target operand. The operation of AJMP and LJMP are analogous to ACALL and LCALL. The SJMP (short jump) instruction provides for transfers within a 256 byte range centred about the starting address of the next instruction (-128 to +127). The PC-relative short jump facilitates the relocatable code.
- JMP @ A + DPTR performs a jump relative to the DPTR register. The operand in the register A is used as the offset (0–255) to the address in the DPTR register. Thus, the effective destination for a jump can be anywhere in the program memory space. This indirect jump is also useful for implementing N-way branches.

ACALL addr11 (Absolute call)

ACALL unconditionally calls a subroutine located at the indicated address. Since ACALL is a 2-byte instruction, PC is incremented by 2 to point to the next instruction. The destination address is obtained by

successively concatenating the five high-order bits of the incremented PC, op-code bits 7–5, and the second byte of the instruction. The subroutine called must, therefore, start within the same 2K block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

(PC)	\square (PC) + 2
(SP)	\square (SP) + 1
((SP))	\square (PC 7–0)
(SP)	\square (SP) + 1
((SP))	\square (PC15–8)
(PC10–0)	\square page address

LCALL addr16 (Long call)

The LCALL calls a subroutine located at the indicated address. Since LCALL is a 3-byte instruction, PC is incremented by 3 to point to the next instruction. The destination address is mentioned in absolute term in the instruction as addr 16. No flags are affected.

(PC)	\square (PC) + 3
(SP)	\square (SP) + 1
((SP))	\square (PC7–0)
(SP)	\square (SP) + 1
((SP))	\square (PC15–8)
(PC)	\square addr15–0

RET (Return from subroutine)

The RET pops the return address from the stack and loads into the PC. Program execution continues at the resulting address. No flags are affected.

(PC15–8)	\square ((SP))
(SP)	\square (SP) – 1
(PC7–0)	\square ((SP))
(SP)	\square (SP) – 1

RETI (Return from interrupt)

The RETI pops the return address from the stack and loads into the PC and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. Program execution continues at the resulting address.

(PC15–8)	\square ((SP))
(SP)	\square (SP) – 1

$(PC7-0) \square ((SP))$
 $(SP) \square (SP) - 1$

AJMP addr11 (Absolute jump)

The AJMP transfers the program execution to the indicated address, which is formed at runtime by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7–5, and the second byte of the instruction. The destination must, therefore, be within the same 2K block of the program memory as the first byte of the instruction following AJMP.

$(PC) \square (PC) + 2$
 $(PC10-0) \square \text{page address}$

LJMP addr16 (Long jump)

The LJMP causes an unconditional branch to the indicated address. No flags are affected.

$(PC) \square \text{addr15-0}$

SJMP rel (Short jump)

Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.

$(PC) \square (PC) + 2$
 $(PC) \square (PC) + \text{rel}$

JMP @A + DPTR (Jump indirect)

The 8-bit unsigned contents of the accumulator are added to the 16-bit data pointer, and the resulting sum is loaded to the program counter. No flags are affected.

$(PC) \square (A) + (\text{DPTR})$

NOP (No operation)

Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

$(PC) \square (PC) + 1$

10.8.2 Conditional Jumps

In the control transfer group, the conditional jumps perform a jump contingent upon a specific condition. The destination will be within a 256 byte range (-128 to +127) centred about the starting address of the next instruction.

The General Format is “Jcond .rel” where rel is the relative address specified. In all the cases, the branch destination is computed by adding the signed relative displacement to the PC, after incrementing the PC to the first byte of the next instruction.

In addition, there are two complex conditional jump instructions.

- The CJNE compares the first operand to the second operand and performs a jump if they are not equal.
- The DJNZ decrements the source operand and returns the result to the operand. A jump is performed if the result is not zero. The DJNZ instruction makes a RAM location efficient for use as a program loop counter by allowing the programmer to decrement and test the counter in a single instruction.

JB bit, rel (Jump if bit set)

If the indicated bit is a 1, jump to the address indicated; otherwise, proceed with the next instruction. The bit tested is not modified. No flags are affected.

$$(\text{PC}) \square (\text{PC}) + 3$$

If (bit) = 1

then

$$(\text{PC}) \square (\text{PC}) + \text{rel}$$

JBC bit, rel (Jump if bit is set and clear bit)

If the indicated bit is a 1, branch to the address indicated; otherwise, proceed with the next instruction. In either case, clear the designated bit. No flags are affected.

Note: When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, not from the input pin.

$$(\text{PC}) \square (\text{PC}) + 3$$

If (bit) = 1

then

$$(\text{bit}) \square 0$$

$(PC) \square (PC) + rel$

JC rel (Jump if carry is set)

If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. No flags are affected.

$(PC) \square (PC) + 2$

If $(C) = 1$

then

$(PC) \square (PC) + rel$

JNB bit, rel (Jump if bit not set)

If the indicated bit is a 0, branch to the indicated address; otherwise proceed with the next instruction. The bit tested is not modified. No flags are affected.

$(PC) \square (PC) + 3$

If $(bit) = 0$

then

$(PC) \square (PC) + rel$

JNC rel (Jump if carry not set)

If the carry flag is a 0, branch to the address indicated; otherwise, proceed with the next instruction. The carry flag is not modified.

$(PC) \square (PC) + 2$

If $(C) = 0$

then

$(PC) \square (PC) + rel$

JNZ rel (Jump if accumulator not zero)

If any bit of the accumulator is a 1, branch to the indicated address; otherwise, proceed with the next instruction. The accumulator is not modified. No flags are affected.

$(PC) \square (PC) + 2$

If $(A) \square 0$

then

$(PC) \square (PC) + rel$

JZ rel (Jump if accumulator zero)

If all the bits of the accumulator are 0, branch to the address indicated; otherwise, proceed with the next instruction. The accumulator is not

modified. No flags are affected.

$$(\text{PC}) \square (\text{PC}) + 2$$

If $(A) = 0$

then

$$(\text{PC}) \square (\text{PC}) + \text{rel}$$

CJNE <dest.byte>, <src.byte>, rel (Compare and jump if not equal)

The CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The carry flag is set, if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected. The first two operands allow four addressing mode combinations—the accumulator may be compared with any directly addressed byte or immediate data and any indirect RAM location, or the working register can be compared with an immediate data.

CJNE A, direct, rel

$$(\text{PC}) \square (\text{PC}) + 3$$

If $(\text{direct}) < (A)$

then $(\text{PC}) \square (\text{PC}) + \text{rel}$ and $(C) \square 0$

or

If $(\text{direct}) > (A)$

then $(\text{PC}) \square (\text{PC}) + \text{rel}$ and $(C) \square 1$

CJNE A, #data, rel

$$(\text{PC}) \square (\text{PC}) + 3$$

If $\#data < (A)$

then $(\text{PC}) \square (\text{PC}) + \text{rel}$ and $(C) \square 0$

or

If $\#data > (A)$

then $(\text{PC}) \square (\text{PC}) + \text{rel}$ and $(C) \square 1$

CJNE Rn, #data, rel

$$(\text{PC}) \square (\text{PC}) + 3$$

If $\#data < (\text{Rn})$

then $(\text{PC}) \square (\text{PC}) + \text{rel}$ and $(C) \square 0$

or

If $\#data > (\text{Rn})$

then $(PC) \leftarrow (PC) + rel$ and $(C) \leftarrow 1$

CJNE @Ri, #data, rel (where Ri = R0 or R1)

$(PC) \leftarrow (PC) + 3$

If $\#data < ((Ri))$

then $(PC) \leftarrow (PC) + rel$ and $(C) \leftarrow 0$

or

If $\#data > ((Ri))$

then $(PC) \leftarrow (PC) + rel$ and $(C) \leftarrow 1$

DJNZ <byte>, <rel-addr> (Decrement and jump if not zero)

The DJNZ decrements by 1 the contents of the location indicated, and branches to the address indicated by the second operand if the resulting value is not a zero. No flags are affected. The location (whose contents are decremented) may be a register or a directly addressed byte.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

DJNZ Rn, rel

$(PC) \leftarrow (PC) + 2$

$(Rn) \leftarrow (Rn) - 1$

If $(Rn) > 0$ or $(Rn) < 0$

then

$(PC) \leftarrow (PC) + rel$

DJNZ direct, rel

$(PC) \leftarrow (PC) + 2$

$(direct) \leftarrow (direct) - 1$

If $(direct) > 0$ or $(direct) < 0$

then

$(PC) \leftarrow (PC) + rel$

Some examples are discussed below.

EXAMPLE 10.7

Produce a low-going output pulse on bit 7 of port 1 lasting exactly 5 cycles.

Solution:

A simple SETB/CLR sequence may be used.

CLR P1.7

NOP

NOP

NOP

NOP

SETB P1.7

One cycle pulse is produced by CLR-SETB and additional 4 clock-cycles are inserted through NOP.

EXAMPLE 10.8

Depending on the value in register A, the following sequence of instructions will branch to one of the four AJMP instructions in a jump table starting at BR_TB.

	MOV	DPTR, #BR_TB
	JMP	@A + DPTR
BR_TB:	AJMP	CODE0
	AJMP	CODE1
	AJMP	CODE2
	AJMP	CODE3

What should be the different values in register A for branching to CODE0 to CODE3?

Solution:

Let us start with $(A) = 0$.

$A + DPTR$ = Address of BR_TB. Thus, $(A) = 0$ will effect branch to CODE0.

–When $(A) = 1$,

$A + DPTR$ = Address of BR_TB + 1. Since AJMP is a 2-byte instruction, the branch will produce undesirable results as it will try to execute the data byte.

–When $(A) = 2$,

$A + DPTR$ = Address of BR_TB + 2. It will effect branch to CODE1.

Similarly, the values of register A must be 4 and 6 for branching to CODE2 and CODE3 respectively.

EXAMPLE 10.9

An array of 10 numbers is stored in the internal data RAM starting from the location 30H. Write a program to move the array starting from location 40H.

Solution:

```

COUNT:    DB      10
          MOV     R0, #30H ; (R0) = 30H
          MOV     R1, #40H ; (R1) = 40H
REPT:    MOV     A, @R0 ; (A) = ((R0))
          MOV     @R1, A ; ((R1)) = (A)
          INC     R0
          INC     R1
;
; Decrement COUNT and jump to REPT if COUNT ≠ 0
;
DJNZ     COUNT, REPT
;
; COUNT = 0. Thus the transfer operation is completed
;
END

```

EXAMPLE 10.10

An array of 20 numbers is stored in the internal data RAM starting from the location 30H. Write a program to search a number NUM in the array and find the number of occurrences of NUM in the array.

Solution:

```

NUM:    DB      -
N-OCCUR: DB      0
COUNT:  DB      20
          MOV     R0, #30H; (R0) = 30H
REPT:   MOV     A, @R0 ; (A) = ((R0))
;
; Compare the array element stored in register A with NUM. Jump to
N-EQUAL
; if (A) ≠ NUM
;
CJNE    A, NUM, N-EQUAL
;
; (A) = NUM. Increment N-OCCUR
;
INC     N-OCCUR
;
; Increment R0 to point to the next array element
;
N-EQUAL: INC     R0

```

```

;
; Decrement COUNT and jump to REPT if COUNT ≠ 0
;
DJNZ      COUNT, REPT
;
; COUNT = 0. Thus the operation on all array elements is completed.
;
END

```

10.9 CONCLUSION

The bit addressing and bit manipulating facility makes the 8051 different from other microprocessors like the 8085. The division and multiplication operations, along with the powerful jump instructions like CJNE and DJNZ, have also added to its capability. We shall see these while discussing case studies in the next chapter.

EXERCISES

1. An array of 10 numbers is stored in the internal data RAM starting from the location 30H. Write a program to find the maximum and the minimum numbers in the array.
2. An array of 20 numbers is stored in the internal data RAM starting from the location 40H. Write a program to
 - (a) sort the array in ascending order.
 - (b) modify the above program for sorting in descending order.
3. Write a program to send 50 output pulses at P2.0. Vary the duration of pulse using NOP.
4. Program the 8051 software timer/counter for time of the day clock. Use three ports to output HRS, MIN and SEC in BCD.
5. An office of public service serves the customers in the sequence of their arrival. The customers take a number tag from the machine and wait in lounge. The employees of the office call customers by pressing a switch on their table which increments and displays the number tag in a two-digit seven-segment display. Each employee also has a built-in counter to count the number of customers handled. Consider that P0.0 to P0.7 are eight lines connected to switches at employees table and port 1 is used to interface the two-digit seven-segment display in serial

mode. Develop a program for the complete system including a software counter for each employee.

6. In a house having five bed rooms, a kitchen, a drawing room and a dining room, the lighting system is to be made automatic in the following way.

1800 HRS	One light each in drawing and dining room is switched on.
1900 HRS	One light in each of five bed rooms is switched on.
2000 HRS	Kitchen light is switched on.
2100 HRS	Kitchen light is switched off.
2200 HRS	Drawing room and dining room lights are switched off.
2300 HRS	Main lights in all rooms are switched off. Night lights in all rooms are switched on.
0600 HRS	Night lights are switched off.

Consider that each light is represented through a port pin. Develop the software for the application.

7. The security system in a bank has 14 hidden switches (known only to the employees) installed at various locations in the bank. When any of the switches is pressed, the location where it occurs is stored and the manager is intimated through the tag number of the location and sounding of an alarm. The manager may, in turn, press a switch to inform the security personnel through an alarm to close the gate and converge to the location of incidence. Consider the various port lines for switches, the alarm and the seven-segment displays at the manager's and security guard's locations. Develop an Intel 8051 software for the application.

FURTHER READING

Douglas, V. Hall, *Microprocessors and Interfacing*, Tata McGraw-Hill, 1999.

Intel Corporation, Santa Clara, *16 Bit Embedded Controller Handbook*.

Intel Corporation, Santa Clara, *Microcontroller User's Manual*.

11

THE 8051 MICROCONTROLLER-BASED SYSTEM DESIGN CASE STUDIES

11.1 INTRODUCTION

Having studied the hardware, the software and the interfacing of the 8051 microcontroller, we shall now take up the task of designing the 8051-based systems for some applications which we see everyday but often overlook. The two applications selected are Traffic Light control and Washing Machine control. These two case studies have been planned to present the systematic view of the 8051-based system design.

11.2 CASE STUDY 1—TRAFFIC LIGHT CONTROL

In this case study, we shall design a traffic light controller using the 8051 microcontroller. Figure 11.1 shows a road cross section. All the roads have a divider strip in between, to facilitate smooth flow of traffic in both directions. The individual roads have been depicted as A, B, C, D, E, F, G and H.

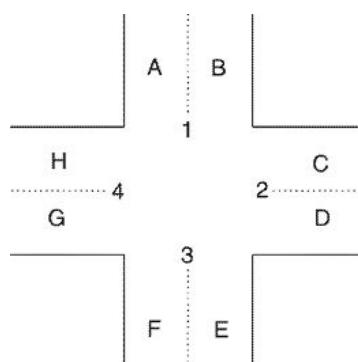
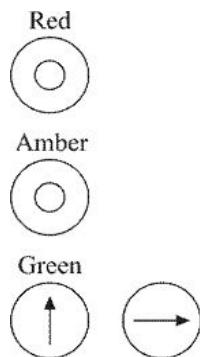


Figure 11.1 A road cross section.

The middle point of each road will have signals for regulating traffic in the forward direction as well as for taking right turns. The left turn is free and uninterrupted. These points have been marked as 1, 2, 3 and 4. Each of these points will contain the following combination of signals.



Let us assign codes to identify the individual signal lights as shown below.

Signal	Traffic points			
	1	2	3	4
Red	1R	2R	3R	4R
Amber	1A	2A	3A	4A
Green(Forward)	1GF	2GF	3GF	4GF
Green(Right)	1GR	2GR	3GR	4GR

If you notice closely, at any middle point, there will be two sets of signals. We shall illustrate this in the following.

Suppose a vehicle is moving on road F towards A, G or C. The driver will first see the signal at point 3, marking R, A, GF or GR. The same signal will be shown at mid-point 1. Thus these signals are duplicated. Similarly when moving on road B, the signals encountered at 1 and 3 will be the same. Thus, the signals at mid-points are the two sets of signals: 1 1□, 2 2□, 3 3□ and 4 4□, as shown in Figure 11.2. However, we may note that:

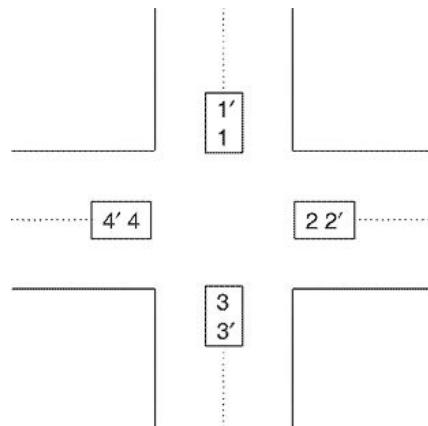


Figure 11.2 Duplicated signals.

- 1 and 3 are same
- 3 and 1 are same
- 2 and 4 are same
- 2 and 4 are same

Since there are only two sets of the same signal conveying the same information placed at different places, we shall treat them as one only in our further treatment.

In addition, for pedestrian traffic on road cross sections, green signals are stationed as shown in Figure 11.3. These are identified as 1GPL, 1GPR, 2GPL, 2GPR, 3GPL, 3GPR and 4GPL, 4GPR. The GPL means left green light for pedestrians and GPR means right green light for pedestrians.

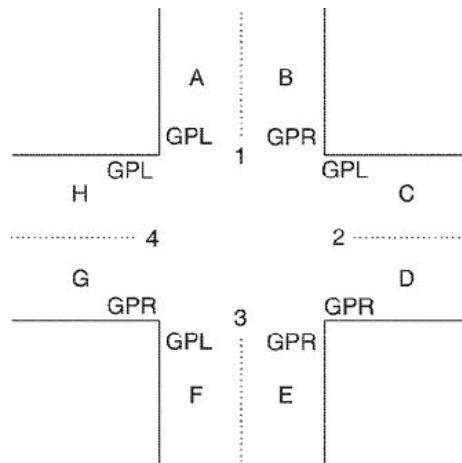


Figure 11.3 Pedestrian traffic control.

Thus, the total number of signal lights to be switched on/off will be

$$\text{At mid-point: } 4 \square 4 = 16$$

$$\text{For pedestrians: } 4 \square 2 = 8$$

$$\text{Total: } = 24$$

The duplicated signals have not been considered as they will get switched on/off as per the main signals.

There can be two strategies for controlling the switching of the signals through the microcontroller.

Strategy1: control each signal light independently

Each light signal is activated individually through a port line using a switching circuit. Thus the total number of port lines required will be 24. The three ports of the 8051 can be effectively used as shown in Figure 11.4.

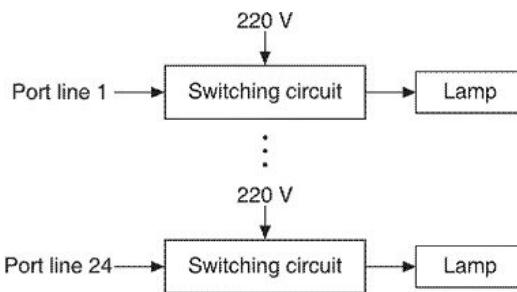


Figure 11.4 Microcontroller interface strategy1.

Strategy2: group the signal lights which are switched on/off together

The following signal lights are switched together.

- GF and GR at middle points
- GPL and GPR at every road cross section

Thus the total number of signal lights requiring individual switching will be

$$\text{At mid-points: } 4 \square 3 = 12$$

$$\text{For pedestrians: } 4 \square 1 = 4$$

$$\text{Total: } = 16$$

Thus, two ports of the 8051 would suffice for switching the signals in this case (Figure 11.5).

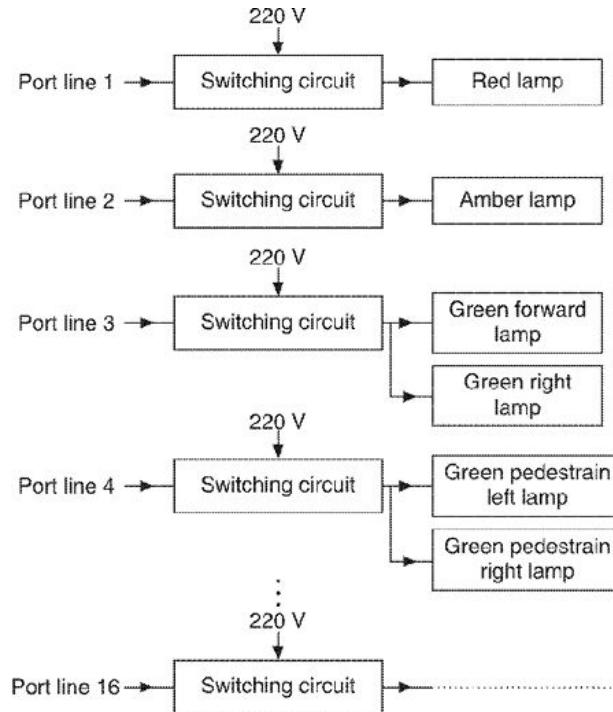


Figure 11.5 Microcontroller interface strategy2.

11.2.1 Switching Circuit

The switching circuit basically connects 220 V to the incandescent lamp on a command from the microcontroller. The switching circuit can be realized using a high power reed relay. Reed relays can be activated by a 5 V signal which is generated at the port pin by writing 1 (Figure 11.6).

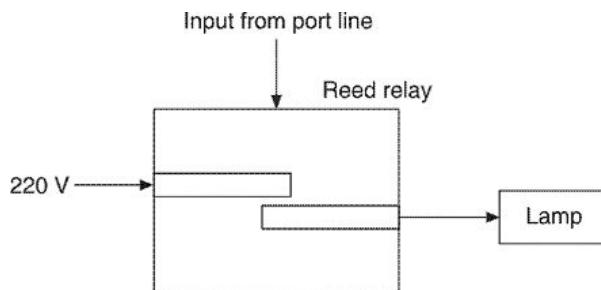


Figure 11.6 Switching circuit based on reed relay.

Alternatively, the same function can be achieved using a TRIAC which comprises two SCRs (Silicon Controlled Rectifiers) joined back to back. The TRIAC can be activated through a 5 V gate signal which can be generated at the port pin (Figure 11.7).

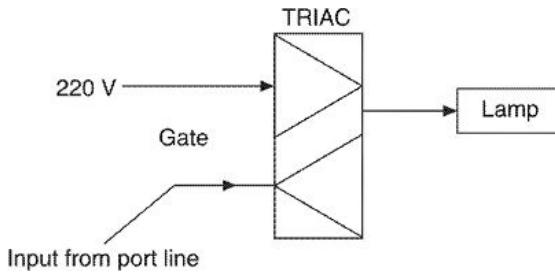


Figure 11.7 Switching circuit based on TRIAC.

11.2.2 The 8051 Hardware Interface

We presume that 4 KB of on-chip ROM will be sufficient to store the program. Thus, all the ports can be used for controlling the traffic signals. Let us now work out the port assignments for different signals.

Port line	Input/output	Assignment
P0.0	Output	1R
P0.1	Output	1A
P0.2	Output	1GF and 1GR
P0.3	Output	1GPL and 1GPR
P0.4	Output	2R
P0.5	Output	2A
P0.6	Output	2GF and 2GR
P0.7	Output	2GPL and 2GPR
P1.0	Output	3R
P1.1	Output	3A
P1.2	Output	3GF and 3GR
P1.3	Output	3GPL and 3GPR
P1.4	Output	4R
P1.5	Output	4A
P1.6	Output	4GF and 4GR
P1.7	Output	4GPL and 4GPR

The 8051 hardware interface is shown in Fig. 11.8.

The switching circuit can be either reed relay based or TRIAC based as explained earlier.

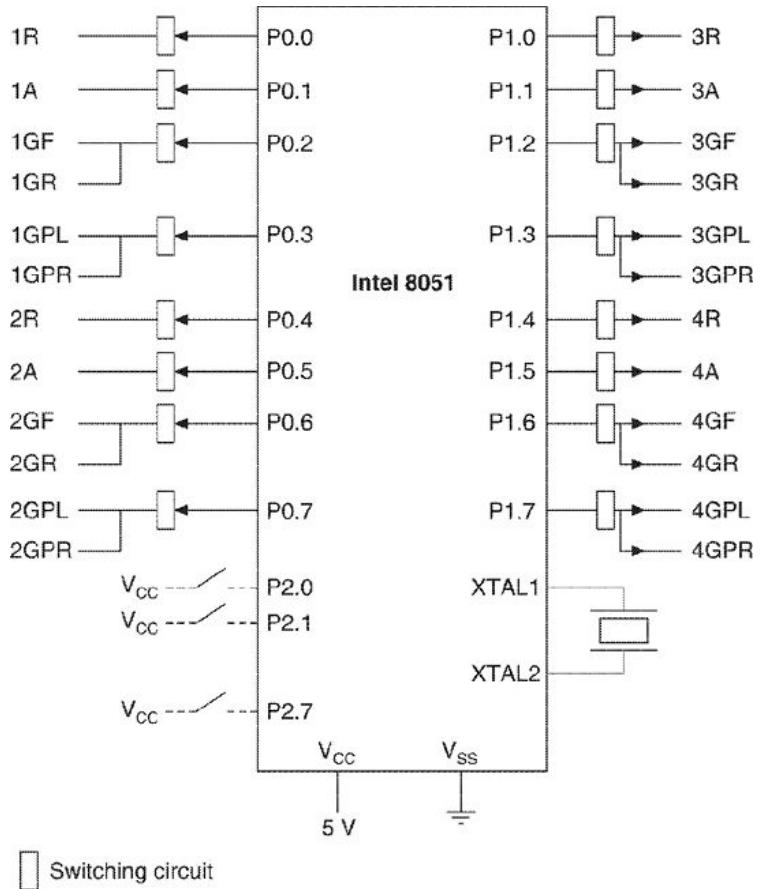


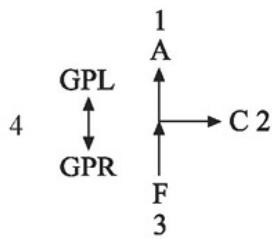
Figure 11.8 Traffic control using the 8051—circuit schematic.

11.2.3 Operation Sequence

Let us now work out the sequence of the operation of the traffic light signals. The traffic density on a road will depend on whether offices/industries are located in the vicinity and also the time of the day. Thus, the green signals for different roads must be put on for different time durations.

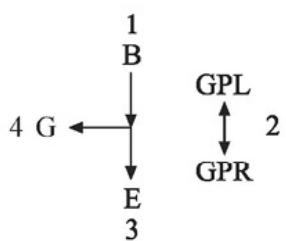
We shall work out the full cycle of operation sequence of signals. After the stage 1 condition, we shall indicate only the signals which change status from the previous stages. The signals, which are not mentioned, are assumed to have status as described in the previous stages.

Stage 1



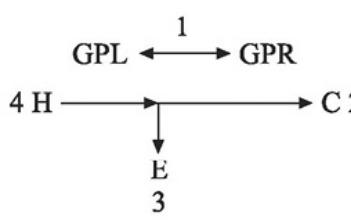
$1A = 2A = 3A = 4A = 0$
 $2R = 3R = 4R = 1$
 $2GF = 2GR = 3GF = 3GR = 4GF = 4GR = 0$
 $1GPL = 1GPR = 2GPL = 2GPR = 3GPL = 3GPR = 0$
 $1R = 0$
 $1GF = 1GR = 1$
 $4GPL = 4GPR = 1$
 Delay1
 $1GF = 1GR = 0$
 $1A = 1$
 $4GPL = 4GPR = 0$
 Delay-A

Stage 2



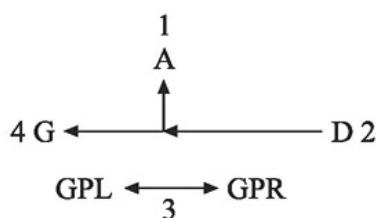
$1A = 0, 1R = 1$
 $3R = 0, 3GF = 3GR = 1$
 $2GPL = 2GPR = 1$
 Delay2
 $3GF = 3GR = 0$
 $3A = 1$
 $2GPL = 2GPR = 0$
 Delay-A

Stage 3



$3A = 0, 3R = 1$
 $2R = 0, 2GF = 2GR = 1$
 $1GPL = 1GPR = 1$
 Delay3
 $2GF = 2GR = 0$
 $2A = 1$
 $1GPL = 1GPR = 0$
 Delay-A

Stage 4



$2A = 0, 2R = 1$
 $4R = 0, 4GF = 4GR = 1$
 $3GPL = 3GPR = 1$
 Delay4
 $4GF = 4GR = 0$
 $4A = 1$
 $3GPL = 3GPR = 0$
 Delay-A

Repeat Stage 1.

Delay-A is the time for which the amber light is put on. Delay1, Delay2, Delay3, Delay4 are the times for which different green lights are

put on. It should be noted that the values of Delay1, Delay2, Delay3 and Delay4 will be different for different roads.

Managing the delay: We consider that these time values can be in multiples of 15 seconds. Thus, Delay1 to Delay4 can be expressed in delay units of 15 seconds. The value of 2 for Delay1 will mean that 1GF, 1GR, 4GPL and 4GPR will be on for 30 seconds. The maximum allowable value of the delay will be 4 units, i.e. 1 minute. The Delay-A, i.e. the time for amber light will be 1 unit, i.e. 15 seconds.

In order to fully utilize the capability of the microcontroller, it is desirable to store the values of Delay1 to Delay4 in the memory in the form of a time-table based on the traffic density observed over a period.

If, for example, offices/industries are located in areas near road A, then the traffic on F \rightarrow A, D \rightarrow A, H \rightarrow A will be heavy in the morning hours from 8:00 am to 10:00 am. The situation in the evening hours from 5:00 pm to 7:00 pm will be opposite to that of the above. To facilitate the commuters, the values of Delay1 and Delay4 can be higher in the morning and that of Delay2 can be higher in the evening. These values can be defined as part of a program and stored in the on-chip ROM of the 8051.

We assume that 18 sets of values of Delay1 to Delay4 are stored in ROM, for regulating traffic from 0600 to 2400 hours. The first set of the delay values will regulate the traffic from 0600 to 0700, and so on. Thus, 72 bytes in total will be used for storing the delay values.

Let us assume that 18 sets of values of Delay1 to Delay4 are stored as part of the program memory starting from location XXXX. Thus,

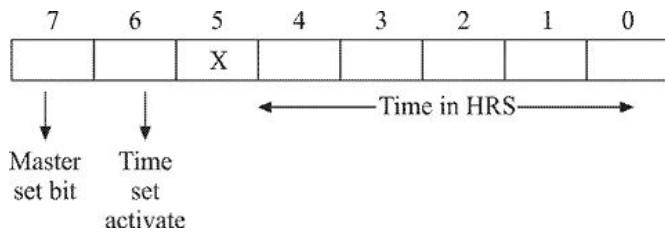
XXXX to XXXX+3 will store Delay1 to Delay4 for 0600 hours.

XXXX+68 to XXXX+71 will store Delay1 to Delay4 for 2300 hours.

11.2.4 Time of the Day Clock

Time of the day clock needs to be designed using the in-built timers/counters to enable the control of the traffic light based on the delay values stored for different hours.

The variables HRS, SEC and MIN denote the current time at any instance. Initially, HRS is set to some value at which the system is started. This can be achieved through switch settings in Port 2. The Port 2 bits have been defined in the following manner.



P2.0 to P2.4 Defines the HRS value of time, i.e. 0 to 24.

P2.7 Master set bit. When = 1, only then time setting will be considered. When = 0, the HRS variable value will be taken for operation.

P2.6 The time setting will be done at the instant the P2.6 becomes 1.

Thus, to set the time to a particular HRS, the value is set in P2.0 to P2.4, and P2.7 is set to 1. At the precise instant when the hour value in user watch equals to the set value in P2.0 to P2.4, P2.6 is set to 1. The microcontroller will modify the HRS variable to a new value, with MIN (minute) and SEC (second) variables being modified to read zero.

The crystal frequency of the 8051 can be from 1.2 MHz to 12 MHz. Since the application in hand is not demanding on time, let us assume that a crystal of frequency 1.2 MHz has been connected. One machine cycle takes 12 oscillator periods, i.e. machine cycle frequency will be 0.1 MHz and each machine cycle will take 10 microseconds. Since the time of the day clock will be continuously running, the most appropriate mode will be mode 2, i.e. Auto Reload. Let us select Timer 1 for realizing the time of the day clock. If the timer is started with an initial value of 56 and automatic reload value stored in TH1 is also 56, then Timer 1 interrupt will be caused after every $200 \times 10 \text{ ms}$, i.e. 2 ms. (*Note:* Timer will start counting 56 onwards and will overflow when the count reaches 256, i.e. after 200 counts.). We shall, therefore, require 500 interrupts to cover a period of one second.

This can be achieved by devising two variables—(IN_COUNT1) Interrupt Counter 1 and (IN_COUNT2) Interrupt Counter 2. As soon as IN_COUNT1 becomes 100, IN_COUNT2 is incremented by 1 and IN_COUNT1 is reset.

As soon as IN_COUNT2 becomes 5, the variable SEC (for seconds) is incremented by 1 and IN_COUNT2 is reset to zero. As soon as SEC becomes 60, the variable MIN (for minute) is incremented by 1 and SEC is reset to zero. As soon as MIN becomes 60, the HRS is incremented by 1 and MIN is reset to zero. As soon as HRS becomes 24, the HRS is reset to zero.

All the above variables, i.e. IN_COUNT1, IN_COUNT2, SEC, MIN and HRS are software counters in the Timer Interrupt ISR.

As soon as the HRS value is incremented by 1, the values of Delay1 to Delay4 stored for the value of Hour, are transferred from memory to register R1 to R4 in register bank 0.

Software

Initialization of the Timer

TMOD SFR

Timer 1, Mode = 02, $C/T = 0$ (Timer Operation) Gate = 0 (Timer Operation triggered by TR1 bit in TCON)

7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0

= 20H

The Timer 1 can be put on by making TR1 bit, i.e. TCON.6 bit as 1.

Delay Subroutine

As explained before, with 1.2 MHz crystal frequency, each machine cycle will take 10 ms. Following instruction

LOOP: DJNZ Rn, LOOP

will take 2 machine cycles, i.e. 20 ms. The maximum value that can be stored in Rn is 255. This means that by placing 255 in Rn and executing the above loop, a delay of 5100 ms, i.e. 5.10 ms can be achieved. For getting 1 second delay, this loop will have to be repeated 196 times.

Subroutine DELAY

; It is assumed that the main program places N as the value of the delay units (Delay unit = ; 15 seconds) in register R0 (Register Bank = 0).

UNIT:	EQU	R6
SCND:	EQU	R5
PSW:	EQU	D0H

; Set Register bank = 0.

MOV	PSW, #00H	
LOOP3:	MOV	UNIT, #0FH; One Delay Unit = 15 seconds
LOOP2:	MOV	SCND, C4H; 1 second = 196 times repetition of loop
LOOP1:	MOV	R7, #FFH

```

LOOP0:      DJNZ      R7, LOOP0
            DJNZ      SCND, LOOP1
; 1 second time delay achieved.

            DJNZ      UNIT, LOOP2
; 1 time unit (15 second) delay achieved.

            DJNZ      R0, LOOP3
; Delay of N time units achieved.

            RET

```

Main Program

IN_COUNT1:	DSB	1
IN_COUNT2:	DSB	1
HRS:	DSB	1
MIN:	DSB	1
SEC:	DSB	1
TEMP1:	DSB	1
IE:	EQU	A8H
TMOD:	EQU	89H
TCON:	EQU	88H
TLI:	EQU	8BH
TH1:	EQU	8DH
	MOV	IN_COUNT1, #00H
	MOV	IN_COUNT2, #00H
	MOV	HRS, #00H
	MOV	MIN, #00H
	MOV	SEC, #00H
; Timer 1 Initialization		
	MOV	A, #20H
	MOV	TMOD, A
; Load the value 56 (38H) in TH1 and TL1.		
	MOV	A, #38H
	MOV	TL1, A
	MOV	TH1, A ; For auto-reload from TH1 to TL1
; Check whether time needs to be set.		

```

CH_TIME:      JB          P2.7, SET_TIME
              SJMP       CH_TIME

; Read the new value.

SET_TIME:     MOV         A, P2
              ANL         A, #1FH; A has the hour value

; Check whether the hour value is equal to or more than 6.

          CJNE        A, #06H, NEQUAL1
          SJMP        NEXT1; (hour value = 6)

; Check for carry flag (hour value < 06).

          NEQUAL1:   JC          ERROR

; Check if HRS is less than 23?

          CJNE        A, #17H, NEQUAL2
          SJMP        NEXT1 (hour value = 23)

; Check for carry flag (HRS < 23).

          NEQUAL2:   JC          NEXT1
          SJMP        ERROR; (hour value > 23)

; Wait till Time set Activate bit = 1.

NEXT1:        JB          P2.6, TIME_SET
              SJMP       NEXT1

; Store the new value.

TIME_SET:    MOV         HRS, A
              MOV         MIN, #00H
              MOV         SEC, #00H

; Start Timer 1.

          SETB        TCON.6

; Enable Timer 1 interrupt by making SFR IE = 10001000B = 88H.

          MOV         A, #88H
          MOV         IE, A

; Load the values of Delay1 to Delay4 from memory to registers R1 to R4.

          ACALL      LD_DELAY

; Execute Stage1 of the signal sequence.

;

; 1R = 0, 1GF = 1GR = 1, 4GPL = 4GPR = 1

```

```

; 2R = 3R = 4R = 1, All other signals = 0
; Port P0 = 0 0 0 1 0 1 0 0 = 14H
;
; Port P1 = 1 0 0 1 0 0 0 1 = 91H
;
;

REPEAT:      MOV          A, #14H
              MOV          P0, A
              MOV          A, #91H
              MOV          P1, A

; Introduce Delay1.

              MOV          A, R1; (R1) = value of Delay1
              MOV          R0, A
              ACALL        DELAY

; Put on amber light 1A and put off 1GF and 1GR, 1GF = 1GR = 0, 1A = 1, 4GPL = 4GPR = 0.

; Port P0 = 0 0 0 1 0 0 1 0 = 12H
;
; Port P1 = 0 0 0 1 0 0 0 1 = 11H
;
;

              MOV          A, #12H
              MOV          P0, A
              MOV          A, #11H
              MOV          P1, A

; Introduce 1 Time Unit Delay.

;
              MOV          R0, 01H
              ACALL        DELAY

; Execute Stage 2 of the signal sequence.
;
; 3R = 0, 3GF = 3GR = 1, 2GPL = 2GPR = 1, 1R = 2R = 4R = 1, All other signals = 0.

; Port P0 = 1 0 0 1 0 0 0 1 = 91H
;
; Port P1 = 0 0 0 1 0 1 0 0 = 14H
;
;

              MOV          A, #91H
              MOV          P0, A

```

```

        MOV      A, #14H
        MOV      P1, A
; Introduce Delay2.
        MOV      A, R2
        MOV      R0, A
        ACALL   DELAY

; Put on amber light.
; ;3A = 1, ;3GF = ;3GR = 0, ;2GPL = ;2GPR = 0, ;1R = ;2R = ;4R = 1, All
other signals = 0.
; Port P0 = 0 0 0 1 0 0 0 1 = 11H
;
; Port P1 = 0 0 0 1 0 0 1 0 = 12H
;
MOV      A, #11H
MOV      P0, A
MOV      A, #12H
MOV      P1, A

; Introduce 1 Time Unit Delay.
        MOV      R0, 01H
        ACALL   DELAY

; Execute Stage 3 of the signal sequence.
;
; ;2R = 0, ;2GF = ;2GR = 1, ;1GPL = ;1GPR = 1, ;1R = ;3R = ;4R = 1, All
other signals = 0.
; Port P0 = 0 1 0 0 1 0 0 1 = 49H
;
; Port P1 = 0 0 0 1 0 0 0 1 = 11H
;
MOV      A, #49H
MOV      P0, A
MOV      A, #11H
MOV      P1, A

; Introduce Delay3.
        MOV      A, R3
        MOV      R0, A
        ACALL   DELAY

; Put on amber light.

```

; 2A = 1, 1R = 3R = 4R = 1, All other signals = 0.

; Port P0 = 0 0 1 0 0 0 0 1 = 21H

;

; Port P1 = 0 0 0 1 0 0 0 1 = 11H

;

MOV	A, #21H
MOV	P0, A
MOV	A, #11H
MOV	P1, A

; Introduce 1 Time Unit Delay.

MOV	R0, #01H
ACALL	DELAY

; Execute Stage 4 of the signal sequence.

;

; 4R = 0, 4GF = 4GR = 1, 3GPL = 3GPR = 1, 1R = 2R = 3R = 1, All other signals = 0.

; Port P0 = 0 0 0 1 0 0 0 1 = 11H

;

; Port P1 = 0 1 0 0 1 0 0 1 = 49H

;

MOV	A, #11H
MOV	P0, A
MOV	A, #49H
MOV	P1, A

; Introduce Delay4.

MOV	A, R4
MOV	R0, A
ACALL	DELAY

; Put on amber light.

; 4A = 1, 1R = 2R = 3R = 1, All other signals = 0.

; Port P0 = 0 0 0 1 0 0 0 1 = 11H

;

; Port P1 = 0 0 1 0 0 0 0 1 = 21H

;

MOV	A, #11H
MOV	P0, A
MOV	A, #21H

```
    MOV      P1, A  
; Introduce 1 Time Unit Delay.  
    MOV      R0, #01H  
    ACALL   DELAY
```

; Repeat the sequence.

```
    SJMP   REPEAT  
ERROR:  NOP
```

Subroutine LD_DELAY

; Subroutine to read and store values of Delay1 to Delay4 from memory depending on ; the value of HRS and load the values of Delay1 to Delay4 in Registers R1 to R4 respectively.

```
XXXX:   EQU      - ; XXXX = Starting address of  
          ; Delay Table in memory  
HRS:    DSB      1  
TEMP1:  DSB      1
```

; Load the starting address of Delay Table in DPTR.

```
    MOV      DPTR, #XXXX
```

; Calculate the Index Value i.e. (HRS-6) \square 4.

```
    MOV      A, HRS  
    SUBB   A, #06H  
    MOV      B, #04H  
    MUL      AB
```

; Since the highest value of HRS is 23, the maximum value of the index will be 68. Thus, the

; result of multiplication will occupy only 8 bits, i.e. only register A.

```
    MOV      TEMP1, A
```

; Load the Delay1 value from memory location XXXX + Index.

```
    MOVC   A, @A + DPTR  
    MOV      R1, A
```

; Load the value of Delay2 to R2.

```
    MOV      A, TEMP1  
    INC      A  
    MOV      TEMP1, A  
    MOVC   A, @A + DPTR
```

MOV R2, A

; Load the value of Delay3 to R3.

MOV A, TEMP1
INC A
MOV TEMP1, A
MOVC A, @A + DPTR
MOV R3, A

; Load the value of Delay4 to R4.

MOV A, TEMP1
INC A
MOVC A, @A + DPTR
MOV R4, A
RET
ISR TIMER 1 Interrupt

IN_COUNT1: DSB 1
IN_COUNT2: DSB 1
SEC: DSB 1
MIN: DSB 1
HRS: DSB 1
IE: EQU A8H

; Increment IN_COUNT1. Check if IN_COUNT1 = 100?

INC IN_COUNT1
MOV A, IN_COUNT1
CJNE A, #64H, NACTION

; IN_COUNT1 = 100, so increment IN_COUNT2 and reset IN_COUNT1.

INC A, IN_COUNT2
MOV IN_COUNT1, #00H

; Check if IN_COUNT2 = 5?

MOV A, IN_COUNT2
CJNE A, #05H, NACTION

; IN_COUNT2 = 5, so increment SEC and reset IN_COUNT2.

INC SEC
MOV IN_COUNT2, #00H

; Check if SEC = 60?

MOV A, SEC

```

        CJNE      A, #3CH, NACTION
; SEC = 60, so increment MIN and reset SEC.
        INC       MIN
        MOV       SEC, #00H
; Check if MIN = 60?
        MOV       A, MIN
        CJNE      A, #3CH, NACTION
; MIN = 60, so increment HRS and reset MIN.
        INC       HRS
        MOV       MIN, #00H
; Check if HRS is equal to or more than 6?
        MOV       A, HRS
        CJNE      A, #06H, STEP1
; HRS = 6
        SJMP      STEP3
; Check if HRS < 06, then no action is required.
        STEP1:   JC       NACTION
; Check if HRS is equal to or less than 23?
        CJNE      A, #17H, STEP2
; HRS = 23
        SJMP      STEP3
; Check if HRS < 23?
        STEP2:   JC       STEP3
        SJMP      NEXT; HRS > 23
; HRS between 6 and 23. So load the new values of Delay1 to Delay4.
        STEP3:   ACALL    LD_DELAY
; Enable Timer 1 interrupt by making SFR IE = 10001000B = 88H.
        MOV       A, #88H
        MOV       IE, A
        RETI
; Check if HRS = 24?
        NEXT:   MOV       A, HRS
        CJNE      A, #18H, NACTION
; HRS = 24, so reset HRS.
        MOV       HRS, #00H

```

; Enable Timer 1 interrupt by making SFR IE = 10001000B = 88H.

```
NACTION:      MOV          A, #88H  
              MOV          IE, A  
              RETI
```

This case study not only illustrates the hardware architecture and software development but also the concept of system design using the 8051 microcontroller.

EXERCISES

1. In the design followed in Case Study 1, the traffic lights work from 0600 to 2400, and are blanked during the remaining hours. Modify the program so that from 0000 to 0600, all the amber lights are made to blink.
2. In the hardware design of Case Study 1, there are no indications for any error. Introduce LED indications for errors in the design and modify the program so that the LEDs glow in case of errors.
3. The design of Case Study 1 has limitation that the system can only be started at a precise hour since there is no provision to enter minutes or seconds value for time of day clock. Replace the existing mechanism of entry of HRS value by a 4 × 4 keyboard, to enable entry of HRS and MIN value. Also introduce an external interrupt in the system design so that the time entry may be done at any point to modify the program.
4. In the program of Exercise 3, also incorporate the facility of entry of the Delay Table using the keyboard.
5. To facilitate the commuters, it is proposed to display the waiting time at all the four traffic signals, through three seven-segment LED displays. The waiting time is displayed in real time, i.e. the delay time which is continuously decreasing every second. Incorporate the seven-segment LEDs and modify the program to display the waiting time in seconds.

11.3 CASE STUDY 2—WASHING MACHINE CONTROL

A top loading automatic washing machine is shown in Figure 11.9. On the top right hand, there are four knobs for machine programming. As we open the top cover, we see a washing basket which can rotate. In the

centre of the basket is a cylindrical vertical column called agitator. The agitator can also move independently. The agitator has a number of vertically pointed fins. The water, detergent and clothes are put in the washing basket. During washing, the agitator and the washing basket rotate in opposite directions in small steps. Due to this action the clothes get washed.

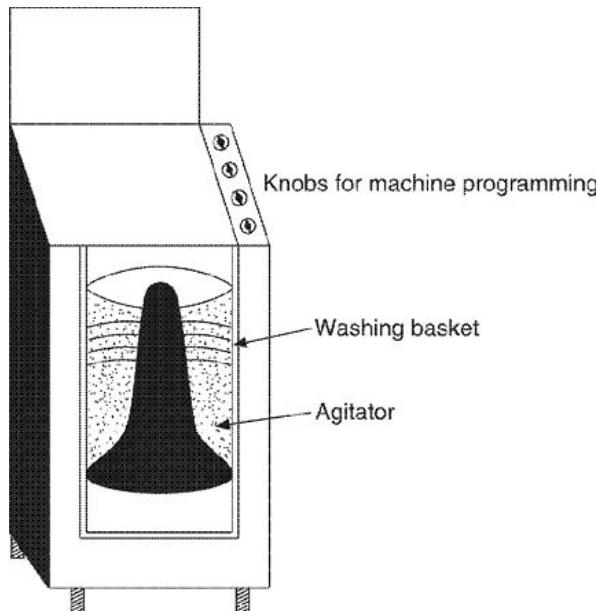


Figure 11.9 Inside view of a washing machine.

Input settings

There are four knobs on the top right-hand side for programming the machine.

Load select: Load basically means the number of clothes intended to be washed together. There are three settings—High, Medium and Low. Based on the load selected, the machine decides the amount of water required.

Water inlet select: Machine can take either hot, tap or mix water. At the back of the machine, there are two inlet pipes for hot and tap water. The knob setting on ‘Mix’ allows 50% tap and 50% hot water as input.

Modes: Through this knob, the machine can be in Normal or Save mode. In the ‘Normal’ mode: (i) the clothes are washed, (ii) the detergent is drained, (iii) the fresh water is put, (iv) the clothes are rinsed, (v) the water is drained, and (vi) using spin the moisture from clothes is taken out to a large extent.

The ‘Save’ mode has been designed to save detergent, and is used when clothes need to be washed in a number of lots. After washing, i.e.

step (i) above, the machine stops. The user may take out the clothes from machine, put another lot and restart the machine from the beginning.

Program select: Using this knob, the machine is programmed to wash the clothes of different kinds. The settings are Extra Heavy, Heavy, Normal, Light and Delicate. Extra Heavy clothes are very dirty clothes like bed sheets, pillow covers, curtains, etc. Heavy clothes are clothes with a lesser dirt level than that of Extra Heavy. Normal clothes are day-to-day personal wears. Light and Delicate clothes are terene and silk clothes. By setting the program select, we program a particular wash cycle. We shall discuss the washing cycle shortly.

Indications

The machine provides the following indications.

Machine on: There is an LED indication which glows when the machine is on.

Washing complete: A sound is generated to announce that the washing is complete.

11.3.1 Washing Cycle

Different operations performed by the machine in a typical wash cycle are shown in Figure 11.10. The figure also describes these operations in different settings of program select, i.e. Extra Heavy, Heavy, Normal, Light and Delicate. The main operations are:

Operation	Extra Heavy	Heavy	Normal	Light	Delicate
Fill	—	—	—	—	—
Agitate	4	—	—	—	—
Soak	6	—	—	—	—
Agitate	4	4	—	8	—
Soak	6	6	—	4	—
Agitate	14	14	14	2	2
Drain	4	4	4	4	4
Spin	4	4	4	4	4
Fill	—	—	—	—	—
Agitate	4	4	4	2	2
Drain	4	4	4	4	4
Spin	10	10	10	4	4

Figure 11.10 Washing machine operation in a typical wash cycle.

Fill: Water is filled through the inlet. The quantity of water depends on the load setting: High, Medium or Low. In the first fill operation, the water temperature is decided by the setting Tap, Hot or Mix. However, in the second fill, i.e. after drain and spin, only tap water is filled for rinsing the clothes. The operation time for fill has not been mentioned as it will be dictated by the quantity of water to be filled and water flow from the tap.

Agitate: In this operation the wash basket rotates in small steps. After every step, there is a two-second wait. Simultaneously, the agitator rotates in the opposite direction in small steps and after every step, there is a two-second wait.

Soak: The operation is basically to allow the clothes to soak the detergent. The machine operation basically stops for a specified time period.

Drain: All the water and detergent are taken out through the drain pipe.

Spin: In this operation, the agitator does not move. The wash basket is rotated at high speed and most of the moisture from clothes is taken out through holes in the inner metallic basket.

11.3.2 Control System Design

With the above knowledge about the design and operation of a washing machine, let us now take up the task of designing the 8051-based control system. We shall assume that the washing machine is under development. Thus, we have the full freedom to plan for electronics and instrumentation components. Let us now analyze our requirements in terms of measurement and control.

Measurement: Quantity of water is being filled.

Control: Following are the control requirements.

- Inlet control of water
- Water quantity control
- Agitator control
- Spin control
- Drain control

Indications: The indications provided in the machine are:

- Machine on indication (LED)

- Washing complete (LED + Buzzer)

Water quantity measurement

The machine has to measure the water quantity during the Fill operation. The measurement schema may be based on the level in the basket or strain caused by the level of the water.

Level measurement

Level measurement transducers working on the principle of float, capacitance and conductivity are available in the market. In this application, we are interested in the information when a particular level has been reached, depending on the load setting. The scheme shown in Figure 11.11 for limit transducer will work for our purpose.

Considering that water is a conductive liquid, electrodes are placed at particular levels in a metal container. When water touches the electrodes, the circuit is completed and a signal is generated.

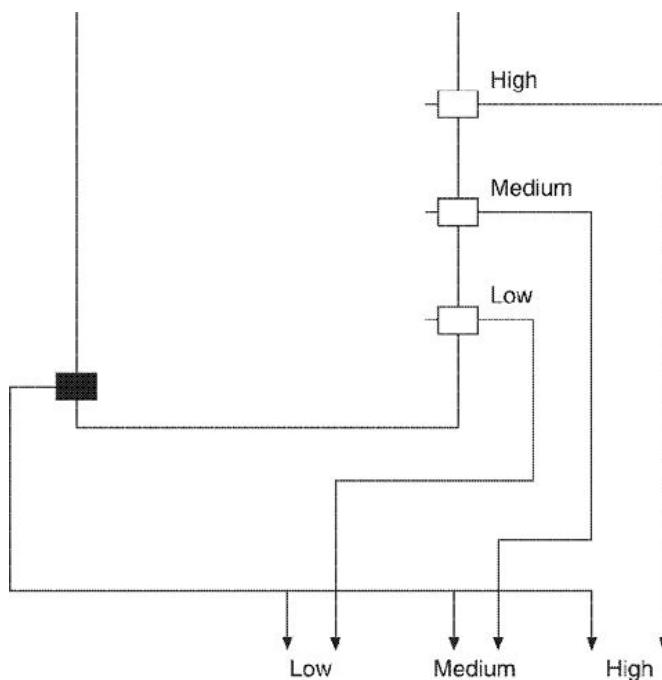


Figure 11.11 Limit transducer for water level measurement.

A silicon liquid-level sensor (ST 004) has also been designed on this principle by Texas Instruments, USA. A tiny chip resides in the sensor with a level wire connected to either face. The ST 004 sensor fits into a probe ST 004B which has holes through which liquid can enter. When the liquid reaches the level of the probe, it enters through the holes and comes in contact with the sensor chip. This changes the temperature of the silicon chip and thereby changes the current passing through the chip.

Strain measurement

Load cells are popular and commonly used as standard transducers for strain measurement. They are basically piezo-resistive transducers.

When a wire is stretched within its elastic limit, it will increase in length and correspondingly the diameter will decrease. Thus, the resistance of the wire changes due to the strain. This is called the *piezo-resistive effect*.

If ΔL be the increase in length and ΔR the increase in resistance, we have

$$\frac{\Delta R}{R} = K \frac{\Delta L}{L} = K \frac{\sigma}{E}$$

where

R = original resistance

L = original length

E = Young's modulus of elasticity

σ = stress = force/area

The value of K varies between 2 and 6 for metals. For semiconductor materials, the values of K up to and above 180 are obtained.

The strain measuring circuit consists of a bridge. One arm of the bridge contains the strain gauge while the other arms have standard resistors of equal resistance as that of gauge resistance. In the unstrained condition, the bridge is balanced and there is no flow of current through the bridge arm. In the strained condition, the current through the bridge arm measures the strain (Figure 11.12).

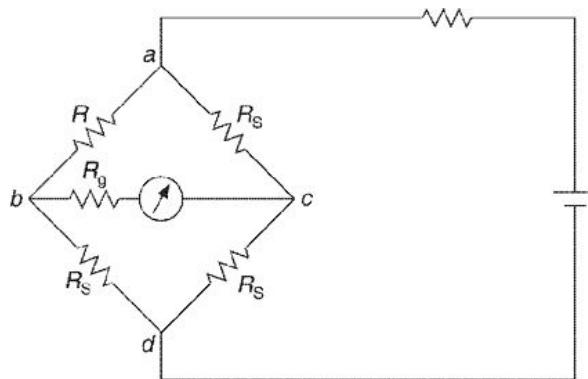


Figure 11.12 Working of strain gauge.

The load cell contains the strain gauge circuit inside it. We may require four load cells to measure the overall strain exerted by the water filled in the basket (Figure 11.13). These will be mounted along the periphery of the bottom of the basket to get a uniform load. The current received by these individual load cells will be summed and compared

with analog current values for high, medium and low water levels. Based on the comparison, the output on high, Medium or Low line may be activated. The analog circuits can be developed using operational amplifiers.

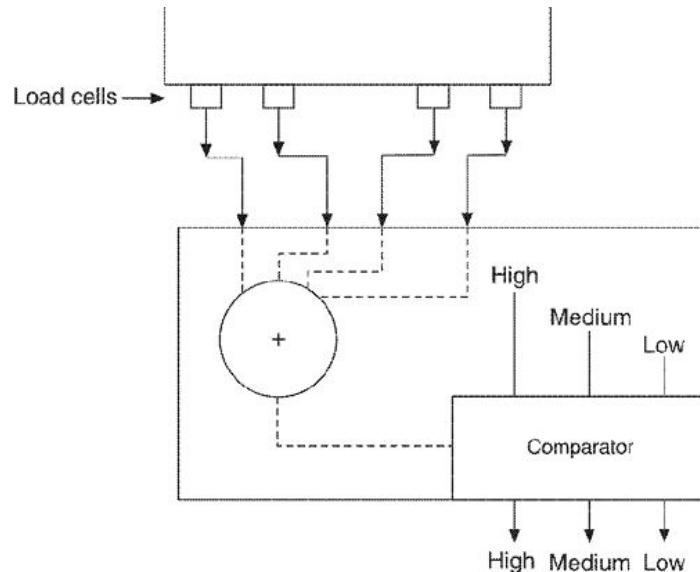


Figure 11.13 Water level measurement using load cells.

Inlet control of water

The inlet water can be tap water, hot water or mix of the two. This is controlled based on the setting done at the water inlet select. The scheme of control is shown in Figure 11.14.

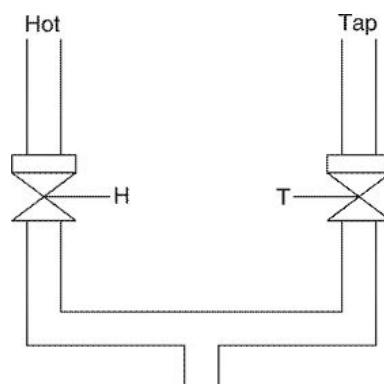


Figure 11.14 Water inlet control.

The hot water and tap water pipes have control valves which enable or disable the flow of water. These control valves are actuated by the control signals H and T as shown in the figure. When $T = 0$ and $H = 1$, only the hot water is allowed; when $T = 1$ and $H = 0$, only the tap water is allowed and when $T = 1$, $H = 1$, mix water is allowed.

Water quantity control

The quantity of water to be filled is dictated by the load selected by the user, i.e. High, Medium or Low. As soon as the water quantity level reaches the desired level, the signal will be received at High, Medium or Low as shown in Figure 11.11 and Figure 11.13. At that instant, the water flow must be disabled by making $H = 0$ and $T = 0$ as shown in Figure 11.14.

Agitator control

During the agitate operation, the agitator moves one rotation in clockwise direction, followed by a rotation in the anticlockwise direction. This cycle is continuously repeated for the specified time. Simultaneously the basket drum undergoes 360° movement, i.e. one full rotation in 1 minute.

Stepper motor control

Both the above movements can be achieved through the stepper motors. Let us call the stepper motors controlling the agitator movement as stepper motor 1 and that controlling the basket drum movement as stepper motor 2.

Stepper motors provide a means for precise positioning and speed control without the use of feedback sensors. The basic operation of a stepper motor allows the shaft to move a precise number of degrees each time a pulse of electricity is sent to the motor. Since the shaft of the motor moves through only the number of degrees that it was designed for when each pulse is delivered, you can control the pulses that are sent and hence control the positioning and speed of the shaft of the motor.

When the stepper motor drive circuitry receives a step pulse, it drives the motor through a precise angle (step) and then stops until the next pulse is received. Consequently, provided that the maximum permissible load is not exceeded, the total angular displacement of the shaft is the step angle multiplied by the number of step pulses received. This relation is further simplified as the shaft position is directly proportional to the number of step pulses supplied, since the step angle for any particular motor is fixed.

The interfacing of stepper motor requires a circuit which can generate the step pulses at the desired rate and the direction signal. A power amplifier is used to amplify these low voltage/power signals to the power required by the motor phases. All this can be done very easily by the microprocessor by using just an output port and simple software, and a power amplifier circuit.

Let us assume that we have a stepper motor with the following characteristics:

1. Four-phase motor, one phase to be activated at a time.
2. Clockwise direction if the phase activation is in the sequences, A, B, C, D, A and anticlockwise if the phase activation sequences is D, C, B, A, D.
3. Voltage at the common point is 24 volts, the current per phase is 1.8 amperes and the step angle is 1.8 degrees.

The block diagram of the microprocessor interface with the stepper motor in shown in Figure 11.15.

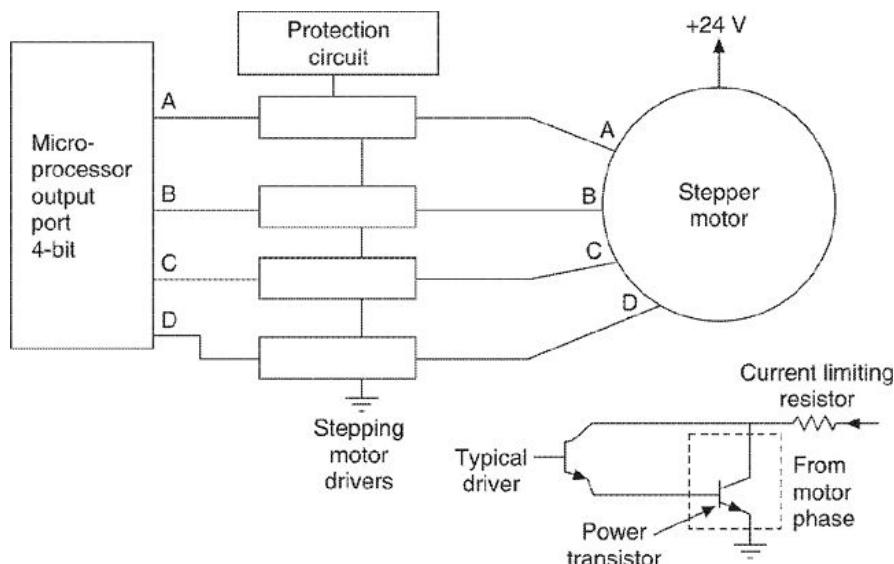


Figure 11.15 Microprocessor interface with stepper motor.

The stepper motor driver circuit is designed specially for a particular motor. A suitable driving transistor (or transistor pair/pairs) capable of providing sufficient gain and desired source/sink current at desired voltage should be selected. It becomes necessary, particularly since a motor of high current type needs to have some protection circuit. It is meant to safeguard the motor and to check that only the desired number of phases are 'ON' and the current passing through the stepper motor coils is not dangerously high (otherwise the motor may burn). This circuit may give a feedback to micropcoessor.

Now, the remaining tasks will be done by the microprocessor. It will generate the phase activation signals (on A, B, C and D lines) on the output port and will wait for the required phase activation time and then depending upon the direction of rotation, it will activate the

corresponding phases. The microprocessor can easily wait for any amount of time starting from few microseconds to even hours. So, it is very easy to control the speed of the motor. Signals required to move the motor in clockwise direction are shown in Figure 11.16.

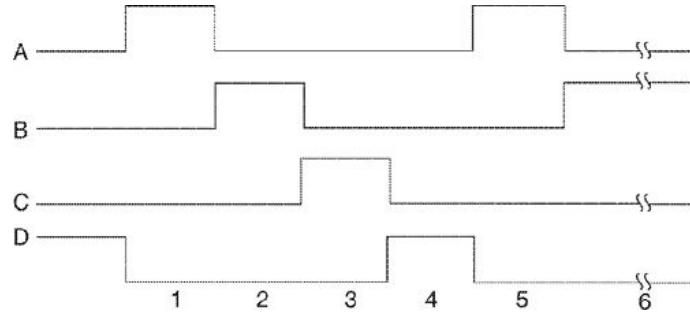


Figure 11.16 Signals for stepper motor movement.

A stepper motor with 1.8 degrees step angle, 4-phase, permanent magnet type has been interfaced to the microprocessor as shown in Figure 11.17. A 4-bit latch IC 7475 is connected

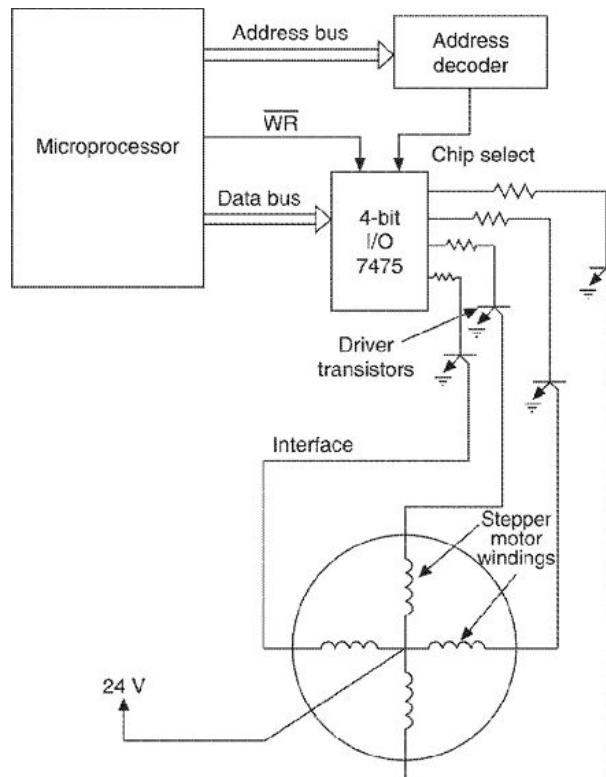


Figure 11.17 Stepper motor control by microprocessor—block diagram.

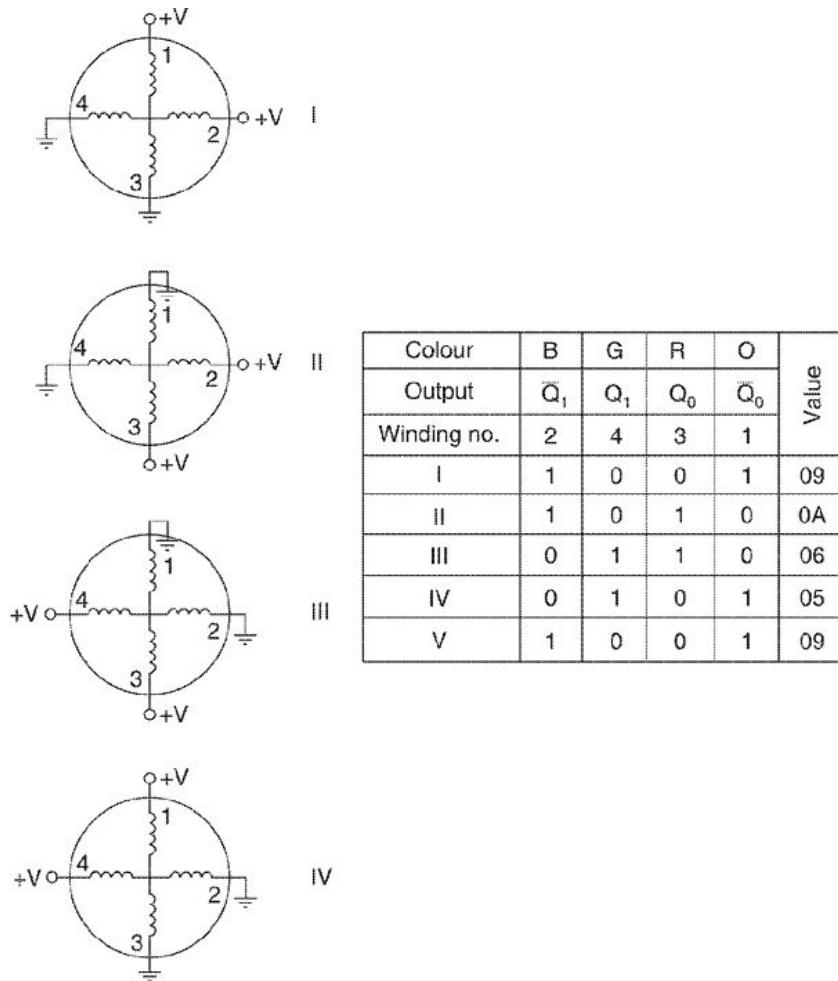


Figure 11.18 Phase activation sequence of stepper motor.

to the data bus of the microprocessor and its output is amplified through transistor 2N 3055. Figure 11.18 shows the phase activation sequence for the full step (1.8 degrees).

Spin control

There is a $\frac{1}{4}$ hp, single-phase, 1400 rpm motor. The control of the spin motor is through a simple on-off control. The control command ($S = 1$) can be an input to the drive circuit which will control the motor in terms of starting or stopping it.

Drain control

For drain control, a control valve is incorporated in the beginning of the drain pipe. The valve can be actuated through the control signal D . When actuated (i.e., when $D = 1$), the control valve allows the water in the basket to flow out through the drain pipe (Figure 11.19).

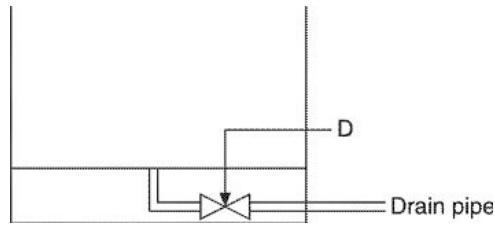


Figure 11.19 Drain control.

Let us now summarize what we have discussed so far. The following signals will carry out the required operations:

Operation	Signal	Type
Program Select	Extra Heavy Heavy Normal Light Delicate	Digital
Water Inlet Select	Hot Tap Mix	Digital
Load Select	High Medium Low	Digital
Measurement of water quantity	High Medium Low	Digital
Water Inlet Control	H(Hot) T(Tap)	Digital
Agitator Control	Stepper Motor 1 (4 Lines) Stepper Motor 2 (4 Lines)	Digital
Spin Control	S	Digital
Drain Control	D	Digital
Indications	Machine on Washing Complete	Digital

We assume that 4 KB of on-chip ROM will be sufficient to store the program. Thus, all the ports can be used for input–output operations. Let us now work out the port assignments for the different input–output signals.

Agitator control requires controlling of both stepper motors 1 and 2. Thus, eight lines will be required for the purpose. The following port assignments are possible:

<i>Port line</i>	<i>Input/Output</i>	<i>Assignments</i>
P0.0	Output	Machine ON Indication
P0.1	Output	Washing complete Indication
P0.2	Output	D—Drain Control Signal
P0.3	Output	H—Hot Water Inlet Control Signal
P0.4	Output	T—Tap Water Inlet Control Signal
P0.5	Output	S—Spin Motor ON-OFF Control Signal
P1.0–P1.3	Output	Stepper Motor 1 Control Signals
P1.4–P1.7	Output	Stepper Motor 2 Control Signals
P2.0	Input	Program Select—Extra-Heavy
P2.1	Input	Program Select—Heavy
P2.2	Input	Program Select—Normal
P2.3	Input	Program Select—Light
P2.4	Input	Program Select—Delicate
P2.5	Input	Load Select—High
P2.6	Input	Load Select—Medium
P2.7	Input	Load Select—Low
P3.0	Input	Water Inlet Select—Hot
P3.1	Input	Water Inlet Select—Tap
P3.2	Input	Water Inlet Select—Mix
P3.3	Input	Water Level—High
P3.4	Input	Water Level—Medium
P3.5	Input	Water Level—Low

It must be emphasized that the above port assignments are random and we may select any port for any function. As an example the stepper motors 1 and 2 control can be assigned to ports 0, 2 or 3 instead of port 1 without any problem. Figure 11.20 shows the washing machine control schematic using the 8051 microcontroller.

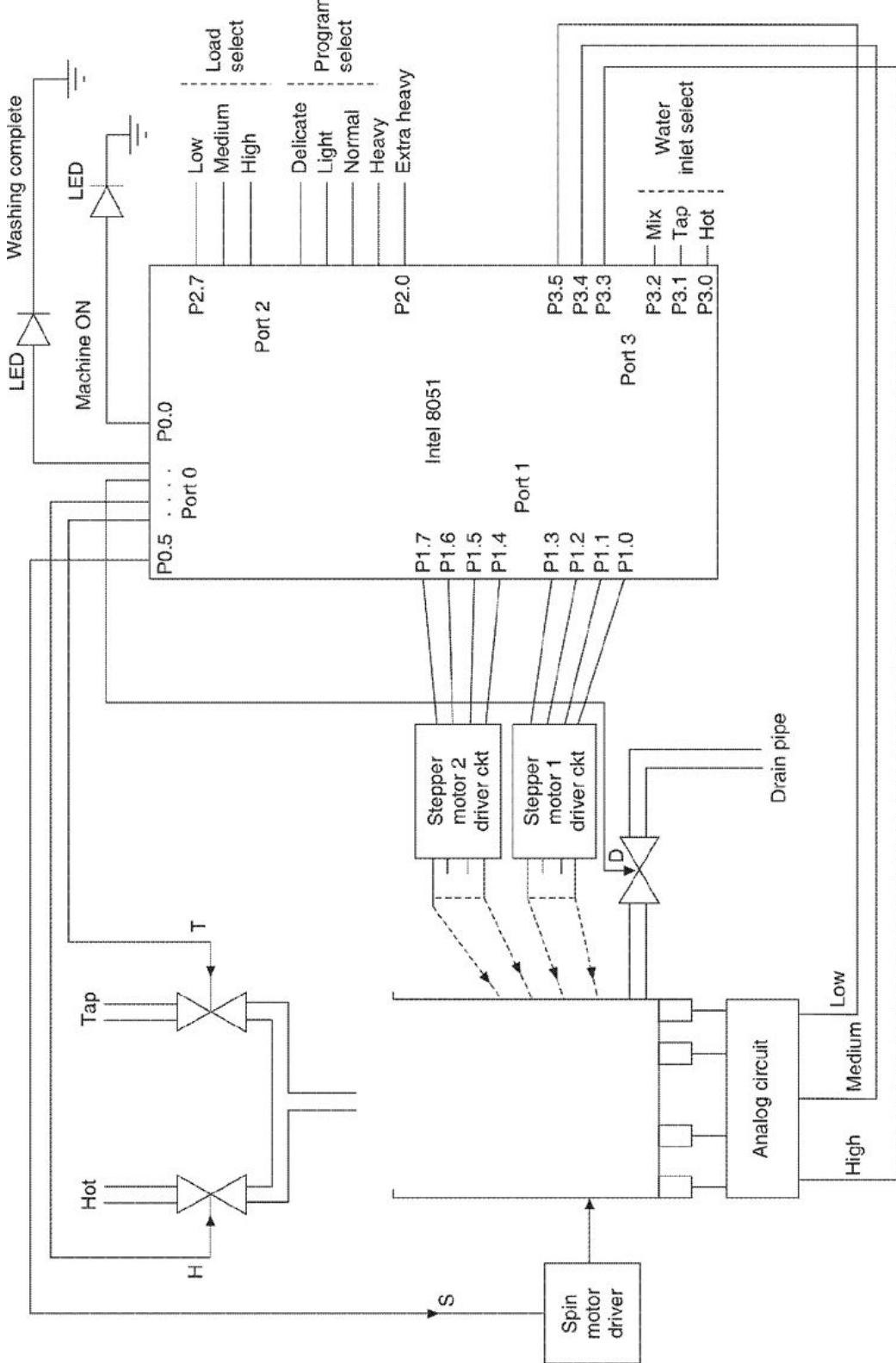


Figure 11.20 Washing machine control using the 8051—circuit schematic.

11.3.3 Software

The task sequence in the form of software modules executed by the washing machine for Extra Heavy Program Select setting is shown in

Figure 11.21. The numbers in parentheses denote the total time of operation. Thus, soak (6) basically means soaking for 6 minutes. As is evident from Figure 11.10, the washing cycles for different program select settings are the subsets of Extra Heavy Washing Cycle.

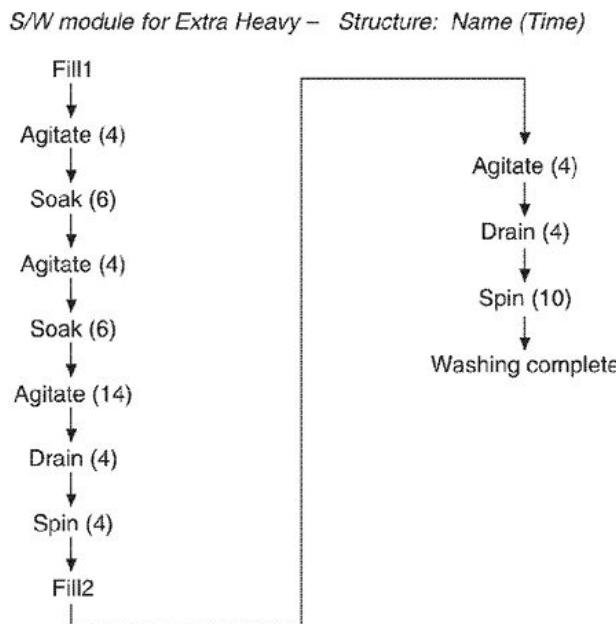


Figure 11.21 Task sequence for Extra Heavy Program Select setting.

We shall develop software for the following modules.

- Fill—Fill1 and Fill2
- Agitate (n)
- Soak (n)
- Drain (n)
- Spin (n)

Let us first develop the **DELAY** subroutine as it will be used in a number of software modules. In order to develop this subroutine, we must assume the basic time reference in the 8051-based system proposed to be used, i.e. clock frequency of the 8051. The crystal frequency can be from 1.2 MHz to 12 MHz. Let us assume 1.2 MHz as the crystal frequency. One machine cycle takes 12 oscillator periods, i.e. the machine cycle frequency will be 0.1 MHz and each machine cycle will take 10 ms.

The following instruction

LOOP: DJNZ Rn, LOOP

will take two cycles, i.e. 20 ms. The maximum value that can be placed in Rn is 255. This means that by placing 255 in Rn, the above loop can cause a delay of 5100 ms, i.e. 5.10 ms. For 1 second delay, the loop has to be repeated 196 times.

Now, let us develop the delay routine.

Subroutine **DELAY**

; It is assumed that the main program places 'N', the value of delay in minutes in register

; R3 (register bank = 0).

PSW:	EQU	D0H
MIN:	EQU	R0
SEC:	EQU	R1

; Set register bank = 0.

MOV PSW, #00H

LOOP3: MOV MIN, #3CH; 1 minute = 60
seconds

LOOP2: MOV SEC, #C4H; 196 times repetition
of loop

; will yield 1 second delay.

LOOP1: MOV R4, #FFH

LOOP0: DJNZ R4, LOOP0

DJNZ SEC, LOOP1

; 1 second time delay achieved.

DJNZ MIN, LOOP2

; 1 minute time delay achieved.

DJNZ R3, LOOP3

; N minute delay achieved.

RET

Subroutine DEL300MRS

; Subroutine to introduce a delay of 300 ms (0.3 ms). This delay is required for

; stepper motor control.

MOV R4, #0FH

LOOP5: DJNZ R4, LOOP5

; 300 ms delay achieved.

RET
Subroutine FILL1

P0: EQU 80H
P2: EQU A0H
P3: EQU B0H

; Check water inlet setting—Hot, Tap, or Mix.

JB P3.0, HOT-W
JB P3.2, MIX-W

; Tap water setting. Send control signal to start the tap water.

SETB P0.4
SJMP CHK_LEVEL

; Hot water setting. Send control signal to start the hot water.

HOT-W: SETB P0.3
SJMP CHK_LEVEL

; Mix water setting. Send control signal to start both hot and tap water.

MIX-W: SETB P0.4
SETB P0.3

; Check load setting—Low, Medium, or High.

CHK_LEVEL: JB P2.7, LOW_LEVEL
JB P2.6, MED_LEVEL

; High level selected. Check whether the desired level is reached.

HIGH_LEVEL: JB P3.3, LEVEL
SJMP HIGH_LEVEL
MED_LEVEL: JB P3.4, LEVEL
SJMP MED_LEVEL
LOW_LEVEL: JB P3.5, LEVEL
SJMP LOW_LEVEL

; Filling complete. Stop water inlet taps.

LEVEL: CLR P0.3
CLR P0.4
RET
Subroutine FILL2

It is same as FILL1 with the difference that only tap water is filled, since water will be used for rinsing.

Subroutine AGITATE (I)

: I = 2, 4, 8 or 14 minutes

; During the agitate operation, the agitator continuously moves one rotation in clockwise

; direction followed by one rotation in anticlockwise direction.

; Simultaneously, the basket drum continuously undergoes rotation in clockwise direction at

; the rate of one rotation in one minute.

: We assume that both agitator and drum motors are four-phase (1000 rpm) stepper motors

; with 1.8 degrees as step angle. One rotation will therefore take 200 steps. Since the drum

; motor has to make one rotation in one minute, the time taken per step will be 0.3 ms. Thus,

; a delay of 0.3 ms has to be inserted between two step movements.

Thus for a 1.8 degree step it will take 0.3 ms.

; Since there are two stepper motors whose operations need to be controlled, there would

; be a requirement of two different mechanisms for introducing the delay. For the basket drum

; stepper motor delay through Timer 1 interrupt has been planned, whereas for the agitator

; stepper motor the software counter has been used to introduce the desired delay.

; We shall now use Timer 1 in Mode 2, i.e. as an 8-bit counter with auto-reload. The SFRs TMOD

; and TCON will be

								←	Timer 1	→	←	Timer 0	→	
7	6	5	4	3	2	1	0							
TMOD	=	0	0	1	0	0	0							

; = 20H

TCON =	0	1	0	0	0	0	0	0
--------	---	---	---	---	---	---	---	---

; = 40H

; Basic timing calculations will be

; The initial count = 226, i.e. E2H. The overflow will

; occur when count = 256, i.e. after $10 \square 30 \text{ ms} = 0.3 \text{ ms}$

PSW:	EQU	D0H
TMOD:	EQU	89H
TCON:	EQU	88H
TL1:	EQU	8BH
TH1:	EQU	8DH
IE:	EQU	A8H
P1:	EQU	90H
OUTLIMIT:	DSB	1
INLIMIT:	DSB	1
LIMIT1:	DSB	1
LIMIT2:	DSB	1

; Set register bank = 0.

MOV PSW, #00H

; Set timer 1 mode = 2, (TMOD = 20H).

MOV A, #E2H
MOV TL1, A
MOV TH1, A
MOV TMOD, #20H
MOV OUTLIMIT, #08H

OUTLOOP: MOV INLIMIT, #D0H; D0H = 208 in decimal

INLOOP: SETB TCON.6; Run Timer 1

; Enable Timer 1 interrupt by making SFR IE = 10001000B = 88H

MOV A, #88H
MOV IE, A

; Stepper motor 2 has been connected to port lines P1.4 to P1.7 (see Figure 11.20). Thus

; for moving stepper motor 2, the sequence code will be output on P1.4 to P1.7 and port

; lines P1.0 to P1.3 will be zero. Thus for clockwise movement, sequence codes values

; 90H, A0H, 60H and 50H will be loaded to P1 in sequence.

; For anticlockwise movement, the sequence code values will be 50H, 60H, A0H and

; 90H.

; Run agitator motor (i.e. stepper motor 2) clockwise 1 rotation (i.e. 200 steps).

MOV	LIMIT1, #C8H
LOOP1: MOV	P1, #90H
MOV	P1, #A0H
MOV	P1, #60H
MOV	P1, #50H

; Introduce 300 ms (0.3 ms) delay to enable stepper motor move in step.

LCALL	DEL300MRS
DJNZ	LIMIT1, LOOP1

; Run agitator motor (stepper motor 2) anticlockwise 1 rotation (200 steps).

MOV	LIMIT2, #C8H
LOOP2: MOV	P1, #50H
MOV	P1, #60H
MOV	P1, #A0H
MOV	P1, #90H

; Introduce 300 ms delay to enable the stepper motor to move in step.

LCALL	DEL300MRS
DJNZ	LIMIT2, LOOP2

; LOOP1 and LOOP2 will take 72 ms each. Thus, a total 144 ms will be taken

; to execute the above. Thus, to continue the agitator operation for 4 minutes, 1666 iterations

; of the above steps will be required. This can be achieved by putting the whole program in

; two loops, one outer loop with 8 counts and one inner loop with 208 counts.

DJNZ	INLIMIT, INLOOP
;	
DJNZ	OUTLIMIT, OUTLOOP
;	
MOV	A, #80H
MOV	IE, A

RET

; Similarly, agitator operations for 14 minutes, 2 minutes and 8 minutes can be achieved
; as required.
;

ISR TIMER 1 Interrupt

IE: EQU A8H

; Stepper motor 1 is connected to ports lines P1.0 to P1.3 (see Figure 11.20). Thus for
; moving stepper motor 1, the sequence code will be output on P1.0 to P1.3 and port lines
; P1.4 to P1.7 will be zero. Thus for clockwise movement sequence code, values 09H,
; 0AH, 06H and 05H will be loaded to P1 in sequence.
; For anticlockwise movement, the sequence code values will be 05H, 06H, 0AH and
; 09H.
; Load code to move stepper motor by 1 step in clockwise direction.

MOV P1, #09H
MOV P1, #0AH
MOV P1, #06H
MOV P1, #05H

; Enable Timer 1 interrupt by making SFR IE = 10001000B = 88H

MOV A, #88H
MOV IE, A
RETI

Subroutine SOAK (6)

P0: EQU 80H

; No machine operations. The clothes are allowed to soak the detergent for 6 minutes.

MOV R3, #06H
ACALL DELAY
RET

Subroutine DRAIN (4)

; Open the drain valve. Wait for 4 minutes.

MOV	R3, #04H
SETB	P0.2
ACALL	DELAY
CLR	P0.2
RET	
Subroutine	SPIN (M)

; M = 4 or 10. Start the spin motor. Insert delay and stop the spin motor.

MOV	R3, #M; (M = 04H or 0AH)
SETB	P0.5
ACALL	DELAY
CLR	P0.5
RET	

Main Program

; Main program initializes all variables and calls all subroutines in sequence.

; Initialize variables.

MIN:	EQU	R0
SEC:	EQU	R1
PSW:	EQU	D0H
P0:	EQU	80H
P1:	EQU	90H
P2:	EQU	A0H
P3:	EQU	B0H
TMOD:	EQU	89H
TCON:	EQU	88H
IE:	EQU	A8H
TL1:	EQU	8BH
TH1:	EQU	8DH

; Put machine on indication.

SETB	P0.0
------	------

; Fill machine with water.

LCALL	FILL1
-------	-------

; Check if Extra Heavy setting?

JNB	P2.0, NEXT
-----	------------

; Extra Heavy setting

LCALL	AGITATE(4)
-------	------------

	LCALL	SOAK(6)
HEAVY:	LCALL	AGITATE(4)
	LCALL	SOAK(6)
NORMAL:	LCALL	AGITATE(14)
	LCALL	DRAIN(4)
	LCALL	SPIN(4)
	LCALL	FILL2
	LCALL	AGITATE(4)
	LCALL	DRAIN(4)
	LCALL	SPIN(10)
	SJMP	DISP_END

; Check for Heavy setting.

NEXT:	JNB	P2.1, NEXT1
	SJMP	HEAVY

; Check for Normal setting.

NEXT1:	JNB	P2.2, NEXT2
	SJMP	NORMAL

; Check for Light setting.

NEXT2:	JNB	P2.3, NEXT3
--------	-----	-------------

; Execute program for Light setting.

	LCALL	AGITATE(8)
	LCALL	SOAK(4)
DELICATE:	LCALL	AGITATE(2)
	LCALL	DRAIN(4)
	LCALL	SPIN(4)
	LCALL	FILL2
	LCALL	AGITATE(2)
	LCALL	DRAIN(4)
	LCALL	SPIN(4)
	SJMP	DISP-END

; Check for Delicate setting.

NEXT3:	JNB	P2.4, STP
	SJMP	DELICATE

; Display washing complete.

DISP-END:	SETB, P0.1
STP:	NOP

END

We have thus been able to develop the 8051-based control for washing machine.

11.4 CONCLUSION

The two case studies dealt with in this chapter depict the applications of the 8051 in day-to-day applications. It is left to you to visualize many more applications where the 8051 can play a prominent role.

EXERCISES

1. The `DELAY` subroutine does not use a timer. Develop the `DELAY` subroutine using TIMER 0, replace the existing subroutine and integrate with the rest of the software. Analyse the benefits derived.
2. The design in Case Study 2 has the crystal frequency as 1.2 MHz. The 8051 can have a crystal frequency up to 12 MHz. Redesign the software assuming 12 MHz crystal frequency. Analyse the benefits derived.
3. The software and the hardware designs in Case Study 2 have not considered the Normal/Save setting. Modify both hardware and software to incorporate this setting.
4. The buzzer output to announce Washing Complete has not been provided in the hardware diagram (Figure 11.20). Develop the buzzer circuit and modify the program to put on the buzzer for 5 seconds.

FURTHER READING

- Allocca, John A. and Allen Stuart, *Transducers: Theory and Applications*, Reston Publishing Company, Pluce, 1994.
- Baumann, H.D., “Trends in control valves and actuator”, *Instruments and Control Systems*, Oct. 1982.
- Considine, Douglas M., *Encyclopedia of Instrumentation and Control*, McGraw-Hill, New York, 1971.
- Fernbaugh A., “Control valves: A decade of change”, *Instruments and Control Systems*, Jan. 1980.

Krishna Kant, *Microprocessor-based Data Acquisition System Design*, Tata McGraw-Hill, New Delhi, 1987.

Krishna Kant, *Computer Based Industrial Control*, Prentice-Hall of India, 1997.

Liptak, B.G., *Instrumentation Engineering Handbook*, Chilton Book Company, Pennsylvania, 1985.

12

INTEL 8096 MICROCONTROLLER HARDWARE ARCHITECTURE

12.1 INTRODUCTION

The 8096 is a 16-bit microcontroller, specially suited for embedded control applications. It has all the features of the 8051, except bit addressing and bit manipulation. Additional features present in the 8096 are A/D converter, high speed inputs, high speed outputs, generation of output analog voltage and mechanism for self-checking in runtime. Coupled with powerful addressing modes and instruction set, these features make the 8096 a powerful microcontroller in wide-spread application environments.

12.2 ARCHITECTURE

The 8096 family of microcontrollers has several sections, all of which work in an integrated fashion to achieve high performance computing and control. The major sections include a 16-bit CPU, a programmable high-speed input/output unit, on-chip RAM, on-chip ROM, analog-to-digital converter, serial port and pulse-width modulated output for digital-to-analog converter. It is available in eight different versions of 48/68 pins.

Following are the resources offered by the 8096 microcontroller.

1. 16-bit CPU
2. On-chip clock generator
3. 64 KB memory space
4. 256 bytes on-chip RAM
5. 8 KB on-chip ROM (*)

6. 26 special function registers
7. 4/8 analog input channels (*)
8. 20/24/40 digital I/O lines (*)
9. 4 high speed inputs
10. 6 high speed outputs
11. 2–16 bits timers/counters
12. 7/8 source interrupt structure
13. 1 D/A output channel
14. Full duplex serial I/O port
15. Watchdog timer

(*) The resources like analog inputs and on-chip ROM are available on certain versions only. There are five (8 bits) I/O ports and all I/O lines are shared on these. These are programmed using special function registers.

The basic facilities offered in different microcontrollers of MCS-96 family are shown in Table 12.1.

Table 12.1 The MCS-96 family of microcontrollers

<i>Micro-controller</i>	<i>Pins</i>	<i>CPU</i>	<i>On-chip RAM</i>	<i>On-chip ROM</i>	<i>Analog channels (*)</i>	<i>Digital I/O lines (*)</i>	<i>High speed input (**)</i>	<i>High speed output (**)</i>	<i>Timer counter</i>	<i>Interrupt source</i>	<i>PWM (D/A output channel)</i>	<i>Serial I/O ports</i>	<i>Watch-dog timer</i>
8096	68	16 bits	256 bytes	Nil	Nil	40	4	6	2	7	1	1	Yes
8094	48	16 bits	256 bytes	Nil	Nil	24	4	6	2	7	1	1	Yes
8396	68	16 bits	256 bytes	8 KB	Nil	40	4	6	2	7	1	1	Yes
8394	48	16 bits	256 bytes	8 KB	Nil	24	4	6	2	7	1	1	Yes
8097	68	16 bits	256 bytes	Nil	8	24	4	6	2	8	1	1	Yes
8095	48	16 bits	256 bytes	Nil	4	20	4	6	2	8	1	1	Yes
8397	68	16 bits	256 bytes	8 KB	8	24	4	6	2	8	1	1	Yes
8395	48	16 bits	256 bytes	8 KB	4	20	4	6	2	8	1	1	Yes

Note:

(*) There are five ports P0, P1, P2, P3 and P4 each of 8 bits.

- Port 0 is shared with eight analog channels both in case of the 8097 and the 8397.
- 4 pins of Port 0 are shared with four analog channels both in case of the 8095 and the 8395.
- Following are not provided on the 48-pin package, i.e. in case of the 8094, the 8394, the 8095 and the 8395.
 - (a) 8 pins of Port 1
 - (b) 4 pins of Port 0 (P0.0–P0.3)
 - (c) 4 pins of Port 2 (P2.3, P2.4, P2.6, P2.7)
- Port 3 and port 4 are shared with address/data lines.
They can be used for interface to external memory and/or I/O.

(**) Two high speed pins HSI.2 and HSI.3 are shared with HSO.4 and HSO.5.

The block diagram of the 8096 is shown in Figure 12.1. Many of the pins are multifunction, i.e. they can perform two different operations under software control. We shall now describe the various building blocks of the 8096 in detail.

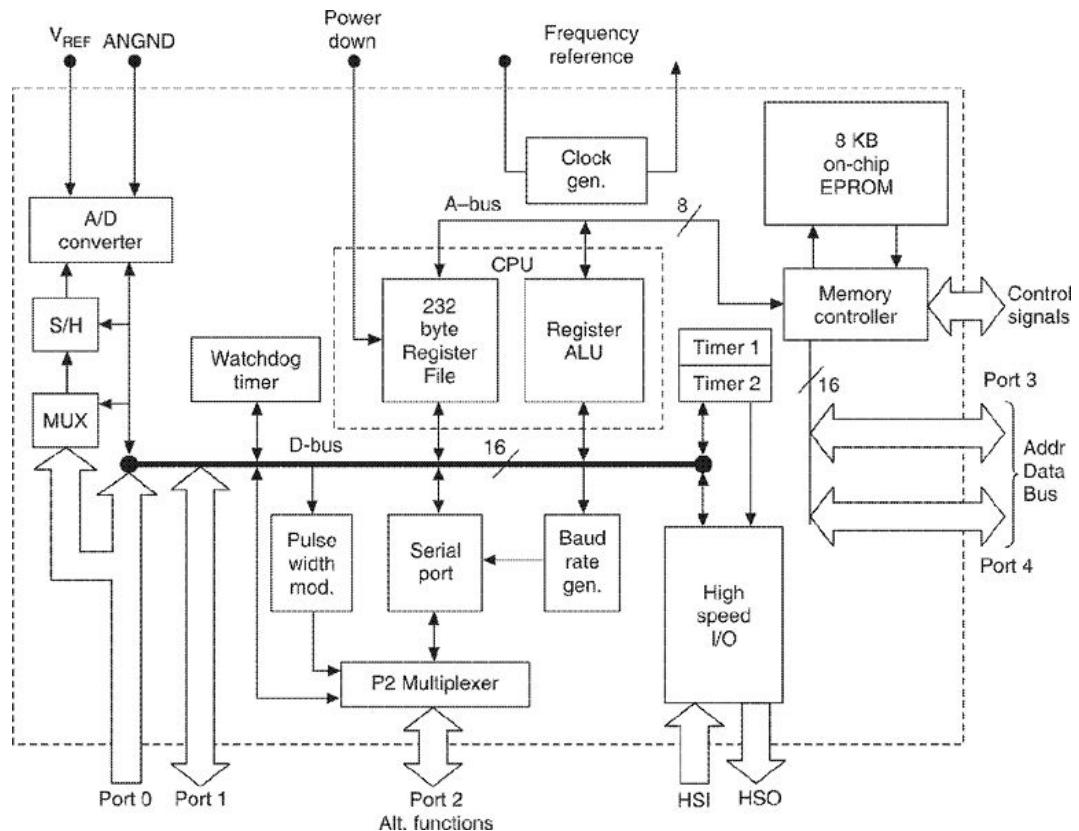


Figure 12.1 Block diagram of the 8096 architecture.

12.3 MEMORY ORGANIZATION

The 8096 can access up to 64 KB of memory. The Scratch Pad Registers (called Register File), Special Function Registers, On-chip RAM, On-chip ROM (in case of the 8396, the 8394, the 8397 and the 8395) and External Memory Space are the main constituents of memory.

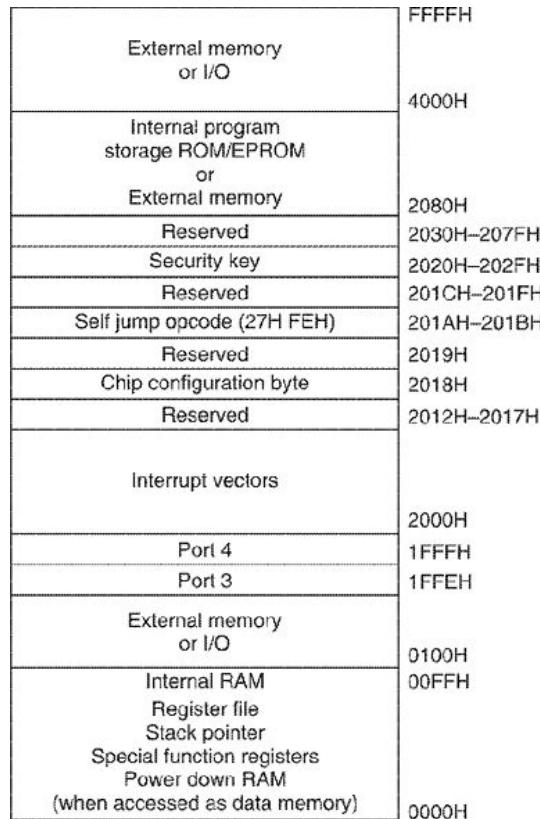


Figure 12.2 Memory map.

Figure 12.2 shows the map of the 64 KB addressable memory space in case of the 8096. The basic blocks are as follows:

(i) Internal RAM containing

- Special function registers (00H to 17H)
- Stack pointer (18H and 19H)
- Register file (1AH to EFH)
- Power down RAM (F0H to FFH)

This memory area is accessed as data memory and no code may be executed from this area. The program memory area of 00 to FFH is reserved for internal use of Intel development systems.

(ii) Internal ROM: In case the chip has on-chip ROM, it would contain

Interrupt vectors

Factory test code

Internal program storage
and be available at addresses (2000H–3FFFH).

If the chip does not contain ROM, then these are defined in the external memory.

(iii) External memory or I/O available at addresses (0100H–1FFDH)
and (4000H–FFFFH)

(iv) Ports 3 and 4 locations (1FFEH and 1FFFH) for reconfiguration,
if these are not used as address/data lines.

When the 8096 is reset, address 2080H is loaded to the Program Counter to give 8 KB of continuous memory.

12.3.1 Central Processing Unit

The Central Processing Unit (CPU) is responsible for performing arithmetic and logic operations and for the generation of control signals. The different control signals are generated depending on the instruction being executed. The CPU of the 8096 contains the following:

- Register File
- Register Arithmetic and Logic Unit (RALU)
- Control Unit

Register file

The complete internal RAM memory map is shown in Figure 12.3. Conforming with many advanced microprocessors, the 8096 does not use the concept of the accumulator. Instead, 230 bytes are reserved in internal RAM (1AH–0FFH) on which the RALU can operate directly. This can be viewed as the 8096 having 230 accumulators. The stack pointer locations 18H and 19H may be used as part of the register file if the stack operation is not performed. This is basically the extension of register bank concept of the 8051.

The upper 16 bytes (0F0H–0FFH) can be kept alive, even when the power fails. This feature is described under Power Down RAM. The area can be used to store critical data about process or system. It must be noted that the register file is only for data storage and no code can be executed from these locations.

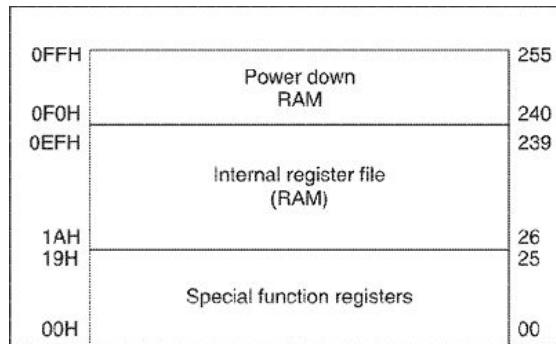


Figure 12.3 Internal RAM memory map.

The CPU communicates with the other resources of the 8096 through Special Function Registers (SFRs) defined in the internal RAM space (00H–19H). Through these SFRs, the CPU controls the various timers, high speed I/O points, interrupts, ADC, stack and I/O ports. It is ultimately the user's program, which commands the CPU to exercise the required control. The SFRs are described separately.

CPU buses

There are two buses A and D (Figure 12.1) for address and data transfers respectively. The different units of CPU, i.e. Register File, RALU and Control Unit interact with each other through these buses. The A-bus is 8-bit wide and is used for the transfer of address information. On the other hand, the D-bus is 16-bit wide and is used for sending/receiving the data information. The reason for making A-bus as 8-bit wide is that the internal on-chip RAM containing SFRs and Register File is 256 bytes long and can be addressed directly using an 8-bit address.

Using these two buses the CPU also communicates with other units of the 8096 as shown in Figure 12.1. The on-chip ROM (in case of the 8396, the 8394, the 8397 and the 8395) and external memory are accessed by the CPU using only A-bus which is multiplexed for address and data. It is evident that two memory cycles will be needed for 16-bit address or data transfer. To minimize such interactions, the memory controller has some built-in facilities which are described under the memory controller.

Register arithmetic and logic unit

The Register Arithmetic and Logic Unit (RALU) contains

- 17-bit ALU
- Program Counter + Incrementer
- Program Status Word

- Loop Counter (5 bit)
- Two Shift Registers (17 bit)
- Temporary Register (17 bit)

The internal structure of RALU is shown in Figure 12.4. The basic philosophy behind providing these many facilities in the RALU is to make it fast and independent. For instructions requiring shift for execution, the shift registers have been provided. The examples of such instructions are Shift Right, Shift Left, Normalize, Multiply, Divide, etc. When a 16-bit quantity is to be shifted, the Upper Word Register/Shifter is used. The Lower Word Register/Shifter is used along with the Upper Word Register Shifter in case of a 32-bit shift.

For instructions requiring repetitive shifts (e.g. ‘Shift left by 5 bits’), a 5-bit loop counter is very useful. For the execution of two operand instructions, a temporary register has been provided. This temporary register stores the multiplier during the execution of the multiplication instruction, divisor during the execution of the division instruction, and so on. For the execution of the increment/decrement instruction, certain constants are defined. The constants 0, 1, 2 are stored in RALU to execute such instructions faster.

Since the A-bus is 8-bit wide and is used to transfer 16-bit address or data information to memory controller or other units, a delay circuit has been provided. It facilitates to transfer the lower byte followed by delay followed by upper byte to the memory controller. The Program Counter and Incrementer are provided in RALU to increment the PC after the execution of each instruction. Thus it points to the next instruction to be executed. In the case of jump instructions being executed, the Program Counter is modified through ALU.

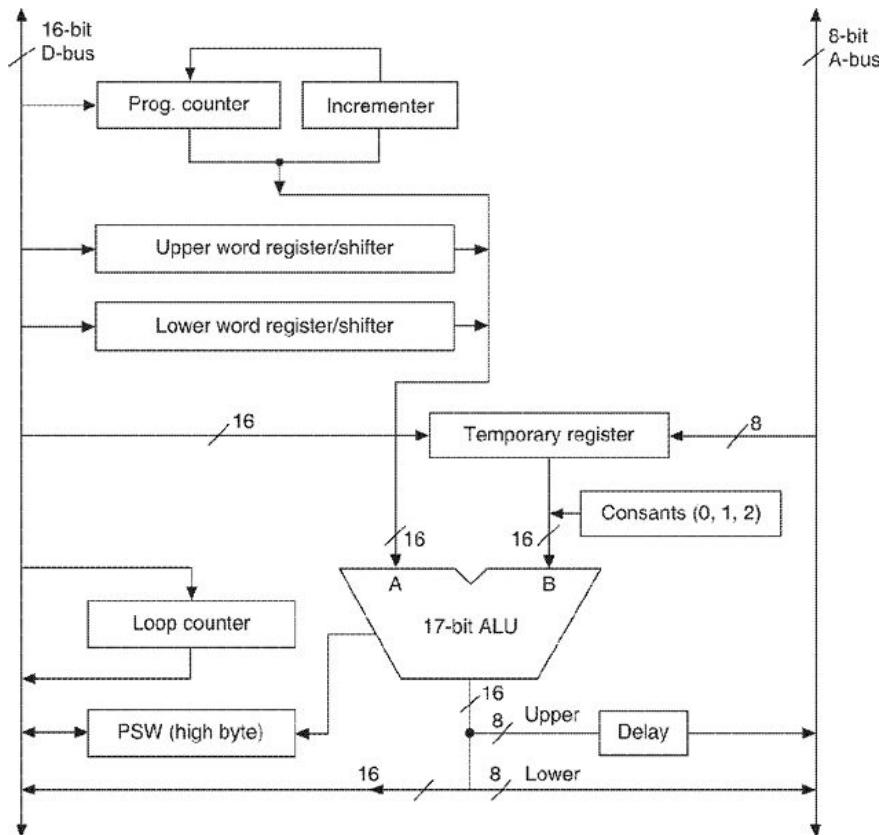


Figure 12.4 RALU block diagram.

Program status word

The Program Status Word (PSW) signifies the status of the interrupt flags as well as the condition flags at any instant (Figure 12.5). The interrupt flags will be described along with the interrupts. The condition flags are set by the execution of the instruction. Following are the flags.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Z	N	V	VT	C	—	I	ST								<Interrupt Mask Register>

Figure 12.5 Program Status Word (PSW).

- Z: Indicates that the result of the last arithmetic/logic instruction was zero.
- N: Indicates that the last instruction generated negative result.
- V: Result generated is outside the range that can be expressed in the destination data type thus causing overflow.
- VT: When the V flag is set, VT (overflow trap) is also set. This can however be reset only by certain explicit instructions. It is useful in debugging the program.
- C: Indicates that a bit is shifted out of MSB or LSB position, because of arithmetic or shift operations.

ST: Can be used for controlling rounding, after right shift. Called ‘sticky bit’, it indicates that 1 has been shifted first to C flag and then out, during right shift.

I: It is set by EI instruction and cleared by DI instruction. It indicates global interrupt enable/disable.

These condition flags (except I) can be used in conditional jump instructions.

Note that PSW here has interrupt status too, along with the status of condition flags. It is different from the 8085, the 8086 and the 8051.

Control unit

The control unit contains the instruction register, the decoder and the timing unit to generate various control signals. The instruction is transferred to the control unit through A-bus and is stored in the instruction register. The instruction is decoded and the required control signals are generated for RALU control.

Memory controller

The memory controller is used as the interface between RALU and external memory or on-chip ROM. The 8-bit wide A-bus is used to transfer the address and data between RALU and memory controller. Whenever RALU wants an instruction/data from memory, it should send the lower byte of address on A-bus, followed by the upper byte of address. The memory controller interacts with the external memory through an external address/data bus AD₀–AD₁₅ (Ports P3 and P4).

Since the external address bus is used for data input/output too, the address is needed to be latched as soon as it is valid. For this purpose, the memory controller sends the ALE (Address Latch Enable) pulse which can be used for latching the address. The memory controller also issues the RD (Read) control signal to the memory for reading. The instruction/data is read from memory at AD₀–AD₁₅ and transferred to RALU in two steps, each involving one byte transfer on A-bus. Thus, the whole process becomes time consuming and also the width of A-bus becomes a bottleneck for achieving systems throughput. To minimize such interactions, the following facilities have been provided in the memory controller.

- Slave program counter
- 3 byte FIFO queue

The slave program counter is incremented after every instruction fetch operation. The memory controller interacts with the external memory and keeps three memory bytes in FIFO ready for RALU. The RALU can take the next instruction/immediate data from FIFO directly. When a jump or a call occurs, RALU loads the slave PC with the new address through the A-bus. This concept is same as that in the 8086.

If the external memory is slower than the 8096, it will take more time to perform the read/write operations. Such devices can use READY line. The READY line can be used to lengthen the external memory cycles, for interfacing to slow dynamic memory or for bus sharing. If the pin is high, the CPU operation continues in a normal manner. If the pin is low prior to the falling edge of CLKOUT, the memory controller goes into a wait mode until the next positive transition in CLKOUT occurs with READY high. The maximum number of wait states that can be entered is decided by bit 4 and bit 5 of CCR (Chip Configuration Register). The CCR is described under Section 12.7 on Memory Interfacing.

When a microcontroller chip has on-chip ROM (addresses 2000H–3FFFH), \overline{EA} is made high. The memory controller on finding $\overline{EA} = 1$, decodes the address, and if the address is between 2000–3FFFH, the instruction fetch is performed from the internal ROM.

\overline{BHE} (Bus High Enable) signal is used to specify to the external memory that the high byte is required to be selected. Since data from the external memory can be either an 8-bit byte or a 16-bit word, it becomes necessary to select either the higher or the lower byte as desired. Following are the conditions applied during this selection.

AD ₀		
0	1	Higher byte
1	0	Lower byte
0	0	Word

12.4 SPECIAL FUNCTION REGISTERS (SFRS)

All the resources in the 8096 are controlled by software. For this purpose, the configuration byte and the command byte are stored in some reserved memory locations identified for different resources. These memory locations are called Special Function Registers. The memory locations 00H to 19H are reserved for this purpose, with upper two bytes being reserved for the stack pointer. In addition to issuing commands, the user may require to know the status of these resources. Therefore, to facilitate this aspect, SFRs serve two functions—one if they are read

from, the other if they are written into. The memory map of SFRs is shown in Figure 12.6. A brief description of SFRs is given in Table 12.2.

19H	STACK POINTER	STACK POINTER
18H		
17H		PWM_CONTROL
16H	IOS1	IOC1
15H	IOS0	IOC0
14H		
13H	RESERVED	RESERVED
12H		
11H	SP_STAT	SP_CON
10H	IO PORT 2	IO PORT 2
0FH	IO PORT 1	IO PORT 1
0EH	IO PORT 0	BAUD_RATE
0DH	TIMER2 (HI)	
0CH	TIMER2 (LO)	RESERVED
0BH	TIMER1 (HI)	
0AH	TIMER1 (LO)	WATCHDOG
09H	INT_PENDING	INT_PENDING
08H	INT_MASK	INT_MASK
07H	SBUF (RX)	SBUF (TX)
06H	HSI_STATUS	HSO_COMMAND
05H	HSI_TIME (HI)	HSO_TIME (HI)
04H	HSI_TIME (LO)	HSO_TIME (LO)
03H	AD_RESULT (HI)	HSI_MODE
02H	AD_RESULT (LO)	AD_COMMAND
01H	R0 (HI)	R0 (HI)
00H	R0 (LO)	R0 (LO)

Figure 12.6 Memory map of SFRs.

Table 12.2 Special function registers

<i>Register</i>	<i>Description</i>
R0	Zero Register—Always reads as a zero, useful for a base when indexing and as a constant for calculations and compares.
AD_RESULT	A/D Result HI/LO—Low and high order results of the A/D converter.
AD_COMMAND	A/D Command Register—Controls the operation of A/D converter.
HSI_MODE	HSI Mode Register—Sets the mode of the High Speed Input unit.
HSI_TIME	HSI Time HI/LO—Contains the time at which the High Speed Input unit was triggered.
HSO_TIME	HSO Time HI/LO—Sets the time for the High Speed Output to execute the command in the Command Register.
HSO_COMMAND	HSO Command Register—Programs what will happen at the time loaded into the HSO Time registers.
HSI_STATUS	HSI Status Registers—Indicates which HSI pins were detected at the time in the HSI

	Time registers and current status of pins.
SBUF (TX)	Transmit buffer for the serial port, holds contents to be outputted.
SBUF (RX)	Receive buffer for the serial port, holds the byte just received by the serial port.
INT_MASK	Interrupt Mask Register—Enables or disables the individual interrupts.
INT_PENDING	Interrupt Pending Register—Indicates that an interrupt signal occurred on one of the sources and has not been serviced.
WATCHDOG	Watchdog Timer Register—Written to periodically to hold off automatic reset every 64K state times.
TIMER1	Timer1 HI/LO—Timer 1 high and low bytes.
TIMER2	Timer2 HI/LO—Timer 2 high and low bytes.
IOPORT0	Port 0 Register—Levels on pins of Port 0.
BAUD_RATE	Register which contains the baud rate. This register is loaded sequentially lower byte first.
IOPORT1	Port 1 Register—Used to read or write to Port 1.
IOPORT2	Port 2 Register—Used to read or write to Port 2.
SP_STAT	Serial Port Status—Indicates the status of the serial port.
SP_CON	Serial Port Control—Used to set the mode of the serial port.
IOS0	I/O Status Register 0—Contains information on the HSO status.
IOS1	I/O Status Register 1—Contains information on the status of the timers and HSI.
IOC0	I/O Control Register 0—Controls the alternate functions of HSI pins.
IOC1	I/O Control Register 1—Controls the alternate functions of Port 2 pins, timer interrupts and HSI interrupts.
PWM_CONTROL	Pulse Width Modulation Control Register—Sets the duration of the PWM pulse.

The MCS-96 chips are available in 48-pin and 68-pin packages, with and without ADC, and with and without on-chip ROM or EPROM. The pin diagrams are shown in Figure 12.7. The 48-pin version is offered in Dual-in-Line Package (DIP), whereas the 68-pin version comes in a Plastic Lead Chip Carrier (PLCC), a Pin Grid Array (PGA), or a Type B Leadless Chip Carrier(LCC).

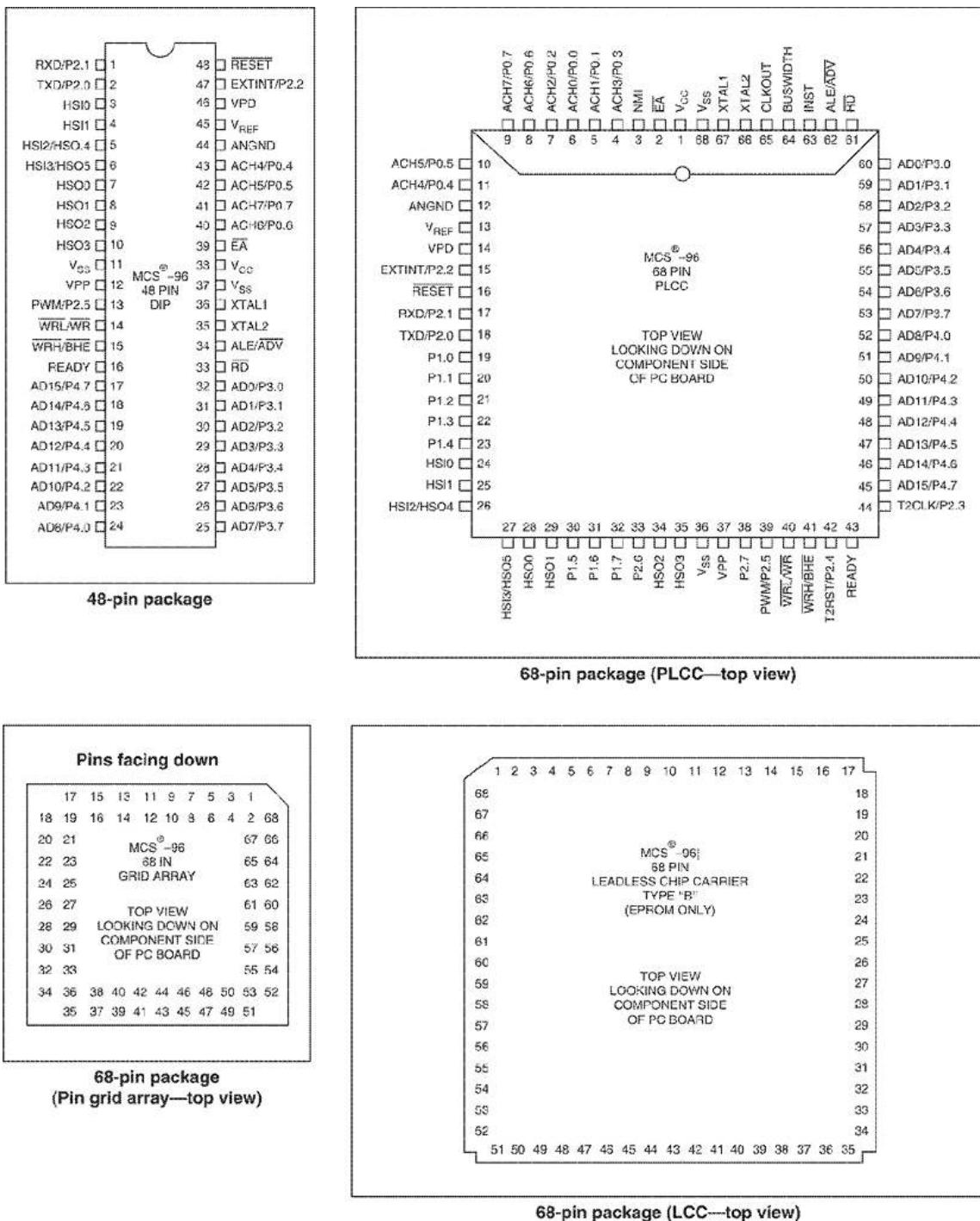


Figure 12.7 The 8096 pin diagram.

12.5 PINS AND SIGNALS

Figure 12.8 shows the pin list of MCS-96 in different packages. These pins and signals are described in Table 12.3.

Name	68-pin PLCC	68-pin PGA	48-pin DIP	Name	68-pin PLCC	68-pin PGA	48-pin DIP
ALE/ $\overline{\text{ADV}}$	62	16	34	P2.3/T2CLK	44	34	—
ANGND	12	66	44	P2.4/T2RST	42	36	—
BUSWIDTH ($\overline{\text{TEST}}$)	64	14	—	P2.5/PWM/ $\overline{\text{PDO}}$	39	39	13
CLKOUT	65	13	—	P2.6	33	45	—
$\overline{\text{EA}}$	2	8	39	P2.7	38	40	—
HSI.0	24	54	3	P3.0/AD0 PVAL	60	18	32
HSI.1	25	53	4	P3.1/AD1 PVAL	59	19	31
HSI.2/HSO.4	26	52	5	P3.2/AD2 PVAL	58	20	30
HSI.3/HSO.5	27	51	6	P3.3/AD3 PVAL	57	21	29
HSO.0	28	50	7	P3.4/AD4 PVAL	56	22	28
HSO.1	29	49	8	P3.5/AD5 PVAL	55	23	27
HSO.2	34	44	9	P3.6/AD6 PVAL	54	24	26
HSO.3	35	43	10	P3.7/AD7 PVAL	53	25	25
INST	63	15	—	P4.0/AD8 PVAL	52	26	24
NMI	3	7	—	P4.1/AD9 PVAL	51	27	23
P0.0/ACH0	6	4	—	P4.2/AD10 PVAL	50	28	22
P0.1/ACH1	5	5	—	P4.3/AD11 PVAL	49	29	21
P0.2/ACH2	7	3	—	P4.4/AD12 PVAL	48	30	20
P0.3/ACH3	4	6	—	P4.5/AD13 PVAL	47	31	19
P0.4/ACH4/MOD.0	11	67	43	P4.6/AD14 PVAL	46	32	18
P0.5/ACH5/MOD.1	10	68	42	P4.7/AD15 PVAL	45	33	17
P0.6/ACH6/MOD.2	8	2	40	$\overline{\text{RD}}$	61	17	33
P0.7/ACH7/MOD.3	9	1	41	READY	43	35	16
P1.0	19	59	—	$\overline{\text{RESET}}$	16	62	48
P1.1	20	58	—	VPP	37	41	12
P1.2	21	57	—	V_{CC}	1	9	38
P1.3	22	56	—	VPD	14	64	46
P1.4	23	55	—	V_{REF}	13	65	45
P1.5	30	48	—	V_{SS}	68	10	11
P1.6	31	47	—	V_{SS}	36	42	37
P1.7	32	46	—	$\overline{\text{WR/WRL}}$	40	38	14
P2.0/TXD/PVER	18	60	2	$\overline{\text{WRH/BHE}}$	41	37	15
P2.1/RXD/PALE	17	61	1	XTAL1	67	11	36
P2.2/EXTINT	15	63	47	XTAL2	66	12	35

Figure 12.8 Pin list of MCS-96.

Table 12.3 Pins and their functions (MCS-96)

Symbol	Name and function
V_{CC}	Main supply voltage (5 V)
V_{SS}	Digital circuit ground (0 V). Two pins.
VPD	RAM standby supply voltage (5 V). This voltage must be present during normal operation. In a power down condition (i.e. when V_{CC} drops to zero), if $\overline{\text{RESET}}$ is activated before V_{CC} drops below specification and VPD continues to be held within specification, Power Down RAM location (16 bytes) from 0F0H to 0FFH of internal RAM will retain their contents. $\overline{\text{RESET}}$ must be held low during the power down and should not be brought high until V_{CC} is within specification and the oscillator has stabilized.
V_{REF}	Reference voltage for A/D converter (5 V). It is also the supply voltage to the analog portion of the A/D converter and the logic used to read Port 0.

ANGND	Reference ground for A/D converter.
VPP	Programming voltage for EPROM parts. It should be +12.75 V when programming and float to 5 V otherwise.
XTAL1	Input of the oscillator inverter and of the internal clock generator.
XTAL2	Output of the oscillator inverter.
CLKOUT	Output of the internal clock generator. The frequency of CLKOUT is 1/3 the oscillator frequency. It has 33% duty cycle.
	Reset input to the chip. Input low for at least two state times to reset the chip.
BUSWIDTH	Input to buswidth selection. If CCR.1 (Chip Configuration Register—see Figure 12.15) = 1, this pin selects the buswidth for the bus cycle in progress. If BUSWIDTH is at 1, a 16-bit bus cycle occurs. If BUSWIDTH is at 0, an 8-bit bus cycle occurs. If CCR.1 is 0, the bus is always an 8-bit bus. If this pin is left unconnected, it rises to V_{CC} .
NMI	A positive transition causes a vector to external memory location 0000H. External memory from 00H to 0FFH is reserved for Intel development systems when accessed as program memory.
INST	Output high during an external memory read indicates that the read is an instruction fetch. INST is valid throughout the bus cycle.
	Input for memory select (External Access). When $\overline{EA} = 1$ the accesses to memory locations 2000H to 3FFFH are directed to on-chip ROM/EPROM. When $\overline{EA} = 0$, the accesses to these locations are directed to off-chip memory. $\overline{EA} = +12.5$ V causes the execution to begin in the Programming mode on EPROM parts. \overline{EA} has internal pull-down, so it goes to 0 unless driven otherwise.
ALE/ \overline{ADV}	Address Latch Enable or Address Valid Output as selected by CCR. Both provide a latch to demultiplex the address from address/data bus. When the pin is \overline{ADV} , it goes inactive high at the end of the bus cycle. \overline{ADV} can also be used as chip select for external memory. ALE \overline{ADV} is activated during external memory accesses.
	Read signal output to external memory. \overline{RD} is activated during external reads. Activated only during external memory reads.
	Write and Write Low output to external memory as selected by CCR. \overline{WR} will go low for every external write whereas \overline{WRL} will go low only when write is at even memory location. Activated only during external memory writes.
	Bus High Enable or Write High Output to external memory as selected by the CCR. $\overline{BHE} = 0$ selects the bank of memory that is connected to the high byte of the data bus. $A_0 = 0$ selects the bank of memory that is connected to the low byte of the data bus. Thus accesses to 16-bit wide memory can be to the low byte (only when $A_0 = 0, \overline{BHE} = 1$), to the high byte (only when $A_0 = 1, \overline{BHE} = 0$) or both bytes (when $A_0 = 0, \overline{BHE} = 0$). If \overline{WRH} function is selected , the pin will go low if the write is at odd memory location.
READY	READY is used to lengthen the external memory cycles, for interfacing to slow dynamic memory or for bus sharing. When READY pin becomes low prior to falling edge of CLKOUT, the memory controller goes into a wait mode, until the next positive transition in CLKOUT occurs. The number of wait cycles permitted is decided by bit 4 and bit 5 of Chip Configuration Register described under Section 12.7.
HSI	Inputs to High Speed Input unit. Four HSI pins are available: HSI.0, HSI.1, HSI.2 and HSI.3. Two of them (HSI.2 and HSI.3) are shared with the HSO unit. The HSI pins are also used as inputs by EPROM parts in the programming mode.
HSO	Outputs from High Speed Output unit. Six HSO pins are available: HSO.0, HSO.1, HSO.2, HSO.3, HSO.4 and HSO.5. Two of them (HSO.4 and HSO.5) are shared with the HSI unit.

Port 0	8-bit high impedance input only port. These pins can be used as digital inputs and/or as analog inputs to the on-chip A/D converter. These pins are also mode inputs to EPROM parts in the programming mode.
Port 1	8-bit quasi bi-directional I/O port.
Port 2	8-bit multi-functional port. Six of the pins are shared with other functions; the remaining two are quasi bi-directional. These pins are also used to input and output control signals on EPROM parts in the programming mode.
Ports 3 and 4	8-bit bi-directional I/O ports with open drain outputs. These pins are also shared with the multiplexed address/data bus, which has strong internal pull-ups. Ports 3 and 4 are also used as a command, address and data path by EPROM parts operating in the programming mode.

12.6 THE 8096 CLOCK

The 8096 provides pins XTAL1 and XTAL2, between which a suitable crystal of the desired frequency may be connected. The input clock frequency of the 8096 should be between 6.0 MHz and 12 MHz. Alternatively, if we have a frequency source, it may be connected to XTAL1 directly. Three internal timing phases are generated (Figure 12.9) by dividing the external frequency by 3. The internal operations are synchronized to either of the phases called phase A, B and C. Phase A is represented externally as CLOCKOUT. These phases repeat every three oscillator periods, collectively referred to as state time.

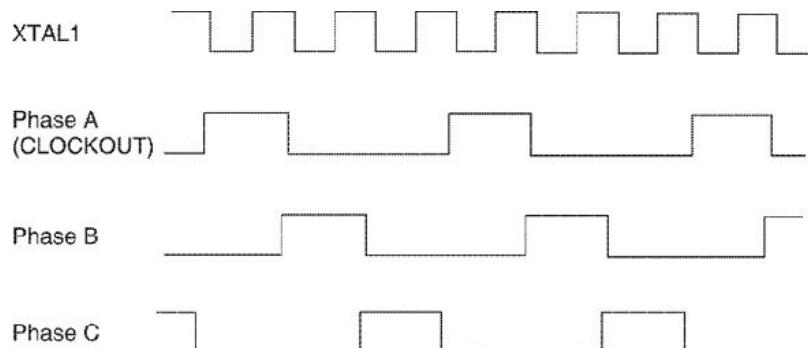


Figure 12.9 Internal timings relative to XTAL1.

The state time is the basic time measurement for any microprocessor. The execution time of every instruction is expressed in terms of number of states. As an example, the EI (Enable Interrupt) instruction takes 4 states to execute. If the external frequency (the frequency of crystal) in the 8096 is 12 MHz, we have

$$\text{Frequency of internal phases} = 4 \text{ MHz}$$

$$\begin{aligned}\text{One period of internal phase} &= 1/(4 \times 10^6) = 250 \text{ ns} \\ &= 0.25 \text{ ms}\end{aligned}$$

Three oscillator periods of the external clock = 1 period of internal phase

= 1 state time

The execution time of EI instruction = $0.25 \times 4 = 1 \text{ ms}$

12.7 MEMORY INTERFACING

To interface a memory chip to the microprocessor, the following signals are required to be generated.

- Address bus signals
- Data bus signals
- Chip select signals
- Read control signal
- Write control signal (only in case of RAM)

Let us now see how these signals are mapped in the 8096 microcontroller environment.

Address and data bus signals are provided through alternate functions in ports P3 and P4 pins.

- Pins P3.0 to P3.7 constitute AD₀ to AD₇, i.e. lower-order address and data lines.
- Pins P4.0 to P4.7 represent AD₈ to AD₁₅, i.e. higher-order address and data lines.
- As in case of the 8085 and the 8086, an external latch will be required to latch the address when present. The latch is strobed through the ALE (Address Latch Enable) signal. The Intel 8282 (described in Chapter 5) latch chip is generally used. The 74LS373 latch chip may also be used.

The chip select signal is obtained by decoding the address information through a decoder. The chip select signal to a memory chip signifies that the address in question is contained in the chip and thus, read or write operation will be performed on the chip selected.

The operations in the 8096 may be performed on bytes as well as words. Thus, a value stored in memory may occupy 8 bits or 16 bits. In the 8096, the word (16 bits) boundary starts at an even address followed

by an odd address. Thus, a word operation at the address 3094H would mean operations on 3094H (lower byte) and 3095H (higher byte).

To facilitate this, the external memory is divided into two address banks—odd address memory and even address memory. The odd address memory bank will have addresses like 0001H, 0003H, and so on, whereas the even address memory bank will have addresses like 0000H, 0002H, 0004H, and so on. Thus, if we wish to interface 4 KB of external memory, we have to physically interface 2 KB as odd address bank and 2 KB as even address bank.

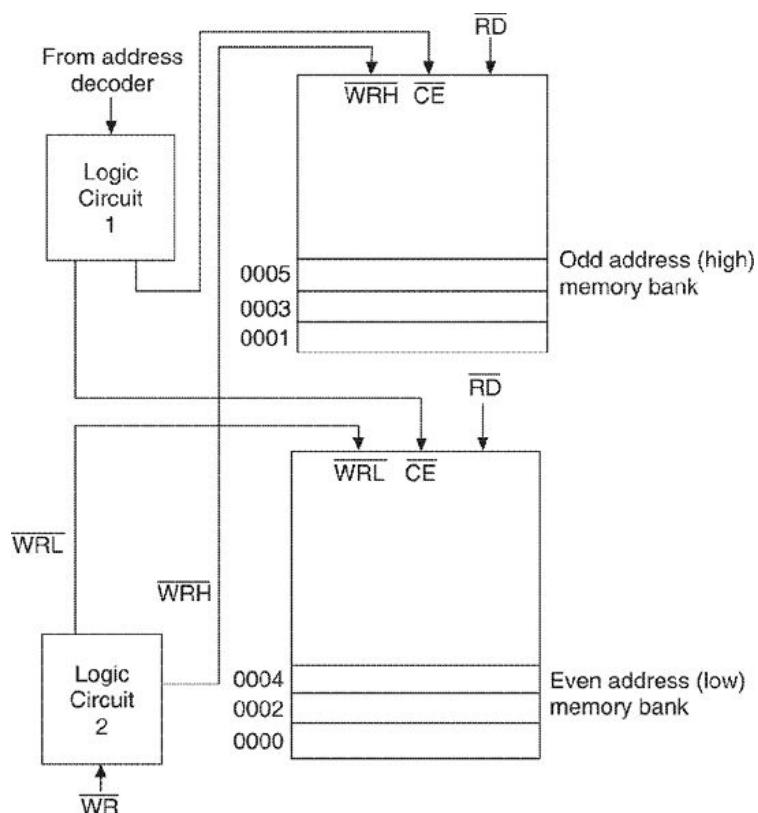


Figure 12.10 High and low memory banks.

Since in a word operation, only the even address is specified in the instruction, which is lower than the odd address, the even address memory bank is also called ‘low memory bank’ and the odd address memory bank is called ‘high memory bank’. Separate chip select signals and read/write control signals will be required for high and low memory banks (Figure 12.10).

- For byte operation, either low or high memory bank will be selected as per the address specified. The read/write control will be performed on the selected bank.

- For word operation, both high and low memory banks will be selected and the read/write will be performed on both the banks.

The output of the address decoder may be further decoded using Logic Circuit 1 to produce chip select for individual memory banks. Similarly the write control signal (\overline{WR}) may be decoded using Logic Circuit 2 to produce \overline{WRH} for high memory bank and \overline{WRL} for low memory bank. It must be noted that it is not required to decode the read control signal (\overline{RD}) as in the 16-bit bus mode, a word is always read starting at an even location. If only one byte is required, then the chip discards the byte it does not need.

The Logic Circuits 1 and 2 can be easily designed using \overline{BHE} and A_0 . These circuits are shown in Figure 12.11 and Figure 12.12. These are the simplest of the circuits to explain concepts. It is possible to design many other circuits for the same operation.

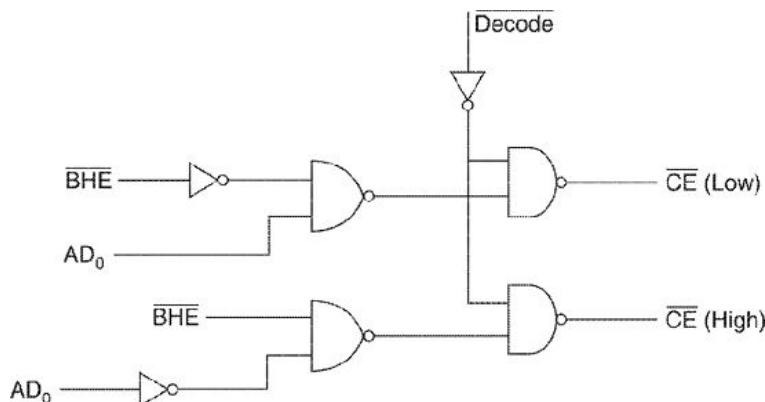


Figure 12.11 Logic Circuit 1.

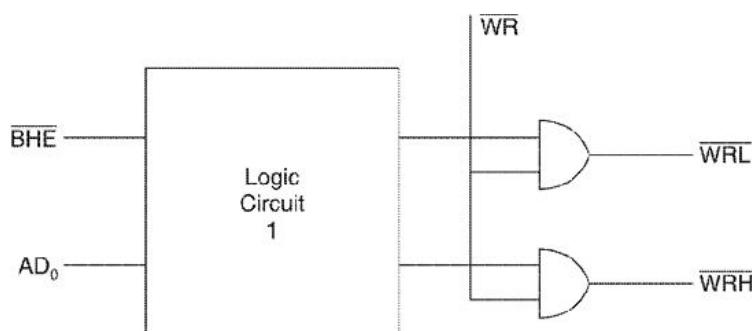


Figure 12.12 Logic Circuit 2.

The address space required for the application is decided, then the same is divided into RAM and ROM. The address space for each chip is decided and based on this, the decoder is connected to different address lines. The 74LS138 decoder is popularly used for address decoding. We shall attempt to interface the following memory chips to the 8096.

Memory chip Address range Memory bank (high/low)

- 2716/6116 (1) 2000H to 2FFFH (2K) Even address (low memory bank)
- 2716/6116 (2) 2000H to 2FFFH (2K) Odd address (high memory bank)
- 2708/6108 (3) 3000H to 37FFH (1K) Even address (low memory bank)
- 2708/6108 (4) 3000H to 37FFH (1K) Odd address (high memory bank)

\overline{RD} , \overline{WR} (or \overline{WRH} and \overline{WRL}) are used as read and write control signals.

The interfacing of the above memory chips to the 8096 is shown in Figure 12.13. The address line A_0 is not connected to the memory chips, but is only used for the selection of low and high memory banks. The lines A_1 to A_{15} are connected to the memory chips depending on the address range. This way the addressing of the consecutive memory locations in the memory banks is made possible. Since Logic Circuit 2 contains (Figure 12.12) Logic Circuit 1, the function of Logic Circuit 2 has been derived from Logic Circuit 1. The 74LS138 decoder used is selected when $G_1 \cdot (\overline{G_2B} \cdot \overline{G_2A}) = 1$. For the address range desired above, A_{15} and A_{14} will be 0 and these are connected to $\overline{G_2B}$ and $\overline{G_2A}$. Following is the truth table of 74LS138.

C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

Following is the logic for generating the chip select signals using the 74LS138 decoder for the memory chips interfaced to the 8096 in Figure 12.13.

Chip	Address range	C B A				
		A_{15}	A_{14}	A_{13}	A_{12}	A_{11}
2716/6116 (2K)	2000H–2FFFH (Even address)	0	0	1	0	0 *
		0	0	1	0	1 *
2716/6116 (2K)	2000H–2FFFH (Odd address)	0	0	1	0	0 *
		0	0	1	0	1 *
2708/6108 (1K)	3000H–37FFH	0	0	1	1	0 *

	(Even address)	0	0	1	1	0	*	*	Y_6
2708/6108 (1K)	3000H-37FFH	0	0	1	1	0	*	*	
	(Odd Address)	0	0	1	1	0	*	*	

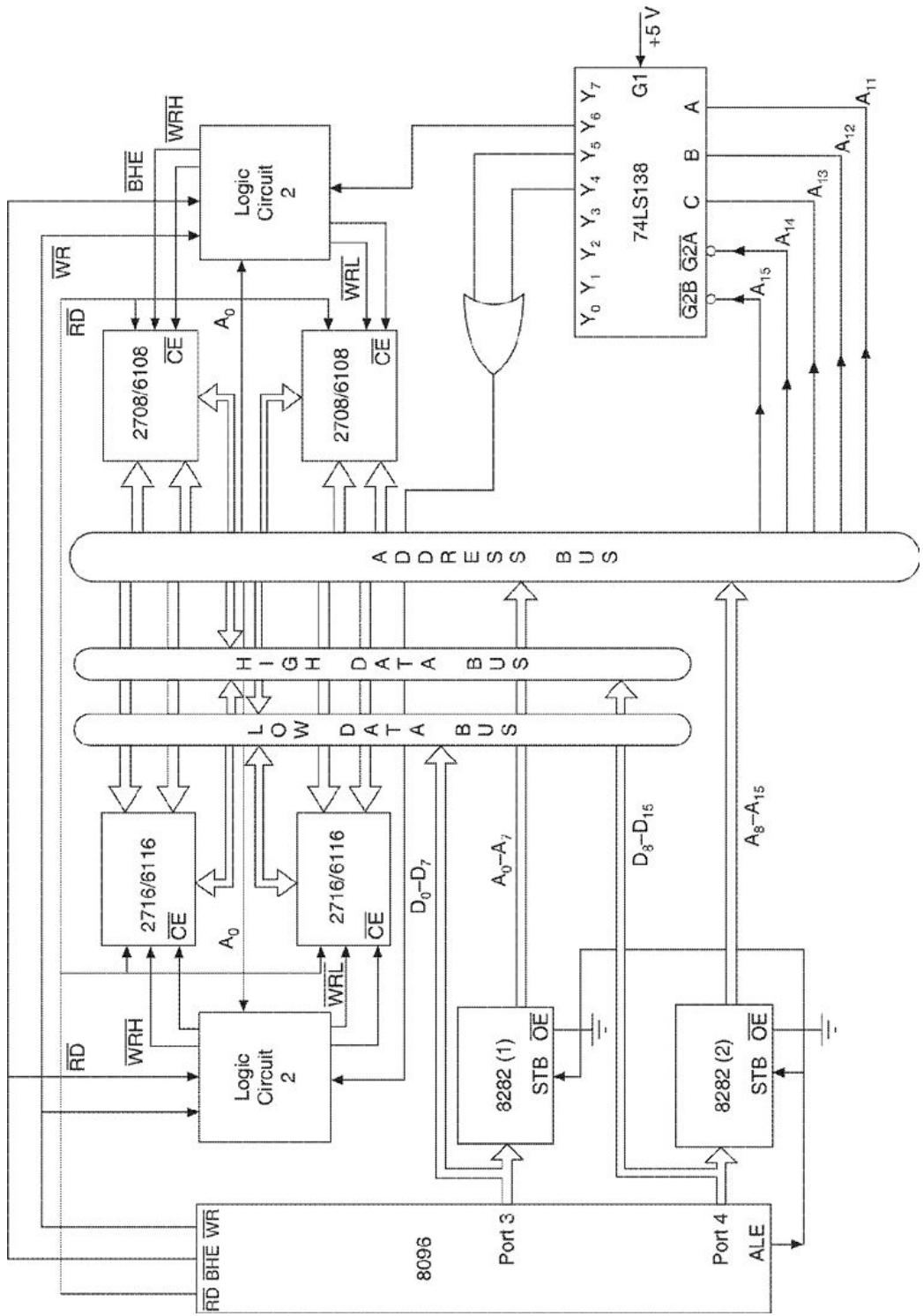


Figure 12.13 Interfacing of memory with the 8096.

The same logic may be extended for interfacing other memory chips. Figure 12.14 shows the external memory timings in case of the standard 16-bit bus described above.

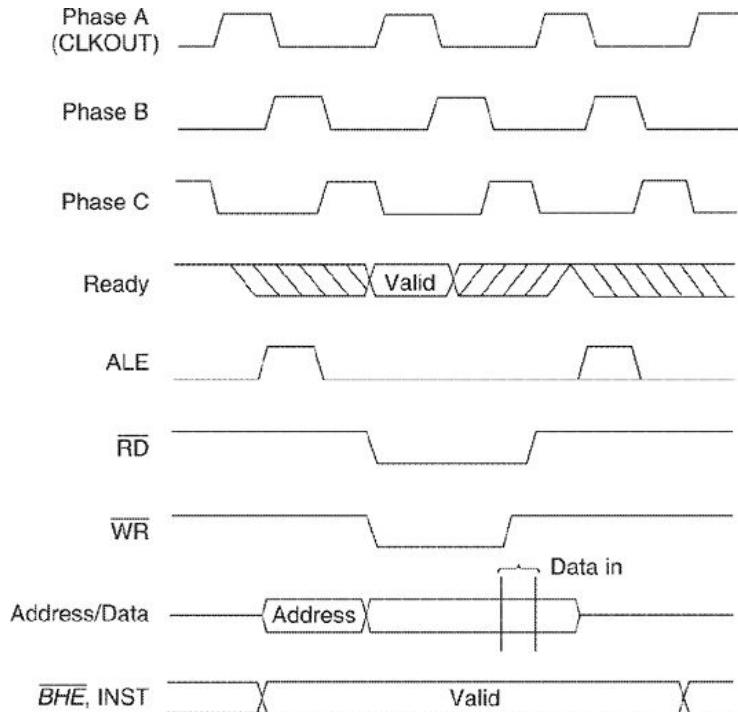


Figure 12.14 External memory timings.

The above interface uses a 16-bit bus cycle, i.e. it is assumed that AD₀ to AD₁₅ may contain 16-bit address as well as 16-bit data in the multiplex fashion. However, there are a large number of applications which require only byte operations. For such applications it would be beneficial if the 8096 memory interfaces are only in 8-bit bus cycle.

In case of 8-bit bus cycles, like in the 8051, the address bus is 16-bit and the data bus is 8-bit wide. The lower 8 bits of the address bus and the 8-bit data bus are time multiplexed. The 8096 microcontroller provides the facility to configure bus width in the runtime and certain other signals for easy interfacing to the external memory through the Chip Configuration Register (CCR).

Configuring the 8096 for memory interfacing

With the aim to enable the designer to easily interface the 8096 to the external memory and to protect the sensitive data/code from intentional or unintentional read/write, the facility to configure the 8096 through the CCR has been provided. The bit map of CCR is shown in Figure 12.15.

The chip configuration byte is located at address 2018H. The CCR is not mapped in memory, but on reset the chip configuration byte is loaded to CCR inside the chip. The 8096, however, reads CCR every bus cycle. Let us now look into the various bits of CCR.

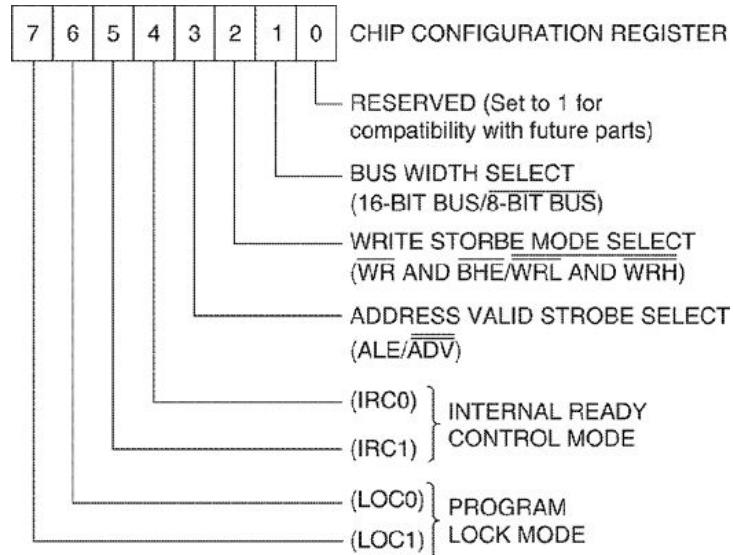


Figure 12.15 Chip configuration register.

Bus width: The external bus width can be configured at runtime to operate as a standard 16-bit multiplexed address/data bus or as a 16-bit address/8-bit data bus (Figure 12.16).

- In the 16-bit bus cycle, ALE is used to multiplex 16-bit address and 16-bit data from AD₀ to AD₁₅ (Port 3 and Port 4).
- In the 8-bit bus cycle, AD₀–AD₇ (Port 3) contain the multiplexed 8-bit data and the lower 8 bits of address, whereas the upper 8 bits of address A₈–A₁₅ are contained in Port 4 and are valid throughout the bus cycle.

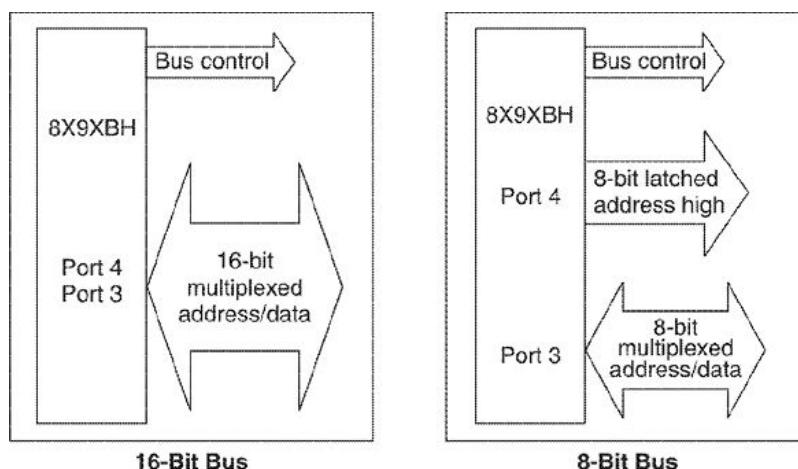


Figure 12.16 Bus width options.

The BUSWIDTH pin of the processor also plays a role in determining the bus width.

BUSWIDTH OR CCR.1 = 0 \square 8-bit bus

BUSWIDTH AND CCR.1 = 1 \square 16-bit bus

Thus, by keeping CCR.1 = 1, and changing the input voltage at the BUSWIDTH pin, the bus width can be changed each bus cycle. It must be, however, understood that some performance degradation does take place when word read or write is performed on the 8-bit bus.

Bus control: The bits 2 and 3 of the chip configuration register can be used to provide the control signals to make the memory interfacing quite easy. If bit 2 = 1, then bus interfacing is through \overline{WR} and \overline{BHE} . It is the Standard Bus Control Mode. Bit 1 = 1 will enable the 16-bit bus whereas bit 1 = 0 will enable the 8-bit bus. The signals used in the bus operation are shown in Figure 12.17. The 16-bit bus interfacing has already been described in Figure 12.13. The 8-bit bus interfacing will not have the data bus D₈–D₁₅ and the latching of higher address bits as these will be valid throughout the bus cycle. All the memory chips will be connected to the same data bus, i.e. D₀–D₇.

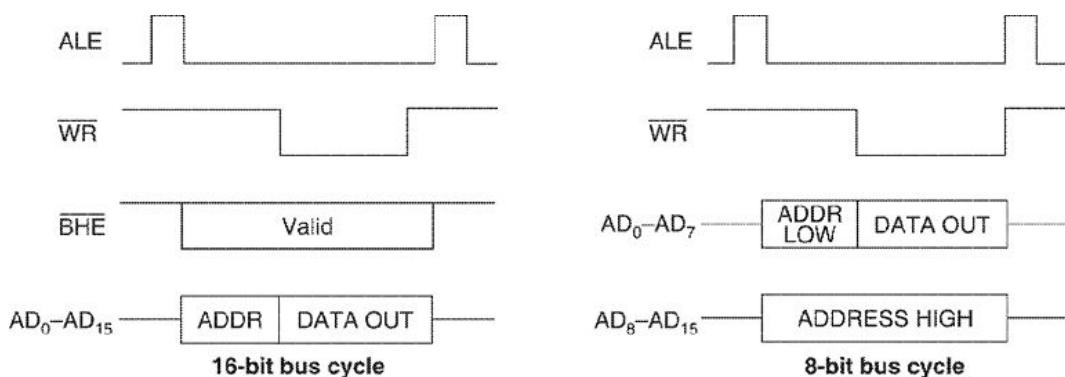


Figure 12.17 Standard bus control.

If bit 2 = 0, then \overline{WRH} (instead of \overline{BHE}) and \overline{WRL} (instead of \overline{WR}) signals are provided. Thus, the external decode to Logic Circuit 2 in Figure 12.13 which produces \overline{WRH} and \overline{WRL} , is avoided. It is called the Write Strobe Mode. The timings for 8-bit bus (CCR.1 = 0) and 16-bit bus (CCR.1 = 1) are shown in Figure 12.18. The external memory interface for 16-bit bus shown in Figure 12.13 will get simplified to Figure 12.19. For the 8-bit bus, 8282 (2) and data bus D₈–D₁₅ will not be there. Port 4 lines will be directly connected to the address bus.

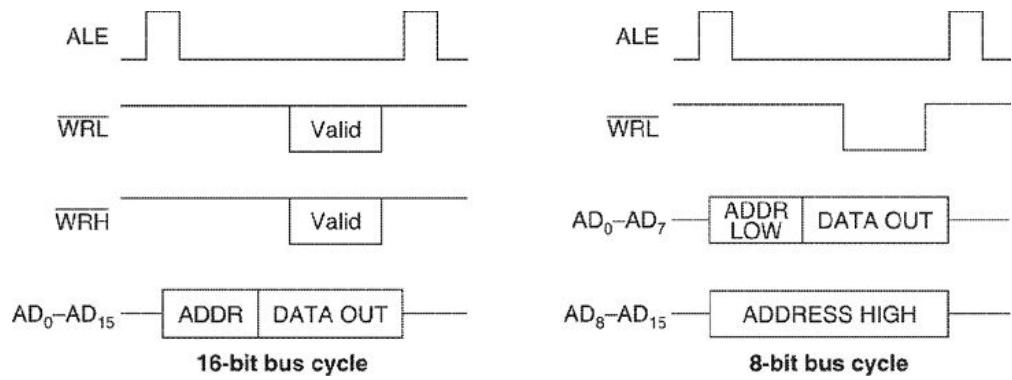


Figure 12.18 Write strobe mode.

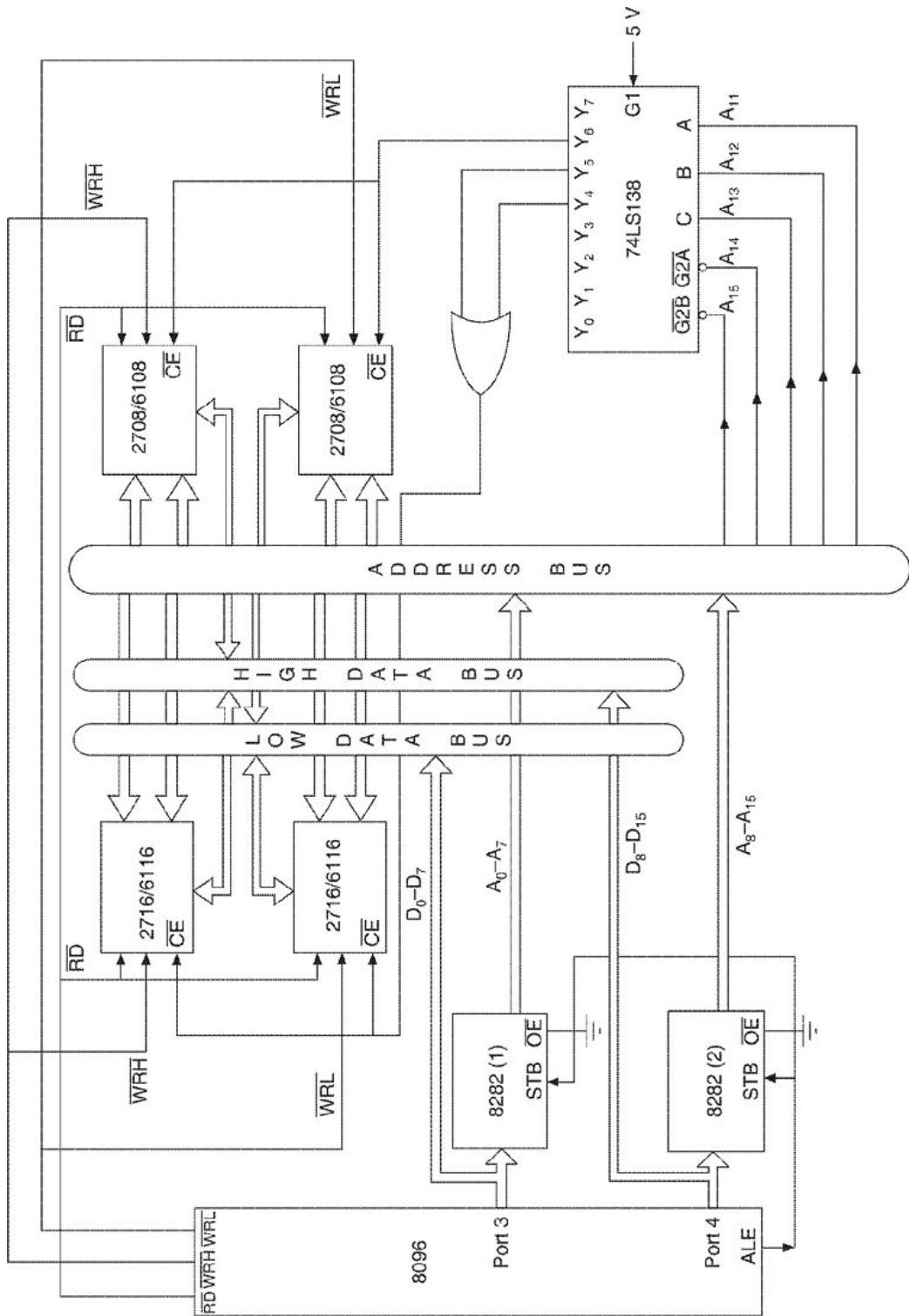


Figure 12.19 Memory interface (16-bit bus cycle) in write strobe mode.

Having avoided Logic Circuit 2, let us examine whether it is possible to avoid address decoding (through 74LS138), also shown in Figure 12.13.

If bit 3 of CCR is 0, then $\overline{\text{ADV}}$ (Address Valid Strobe) is produced in place of the ALE signal. $\overline{\text{ADV}}$ will go low after an external address is set

up and will go high only at the end of the bus cycle. The $\overline{\text{ADV}}$ signal can be used for address latching in place of ALE (address is latched on 1 to 0 transition of ALE) as well as chip select to memory, if only one even and one odd memory banks are interfaced.

The $\overline{\text{ADV}}$ will be directly connected to chip select input of the memory chips. The selection between two banks (even or odd) will be done by $\overline{\text{WRH}}$ and $\overline{\text{WRL}}$. This is known as Address Valid Strobe Mode. Figure 12.20 shows the signals in case of the address valid strobe mode for the 16-bit bus cycle ($\text{CCR.3} = 0$, $\text{CCR.1} = 1$) and the 8-bit bus cycle ($\text{CCR.3} = 0$, $\text{CCR.1} = 0$)

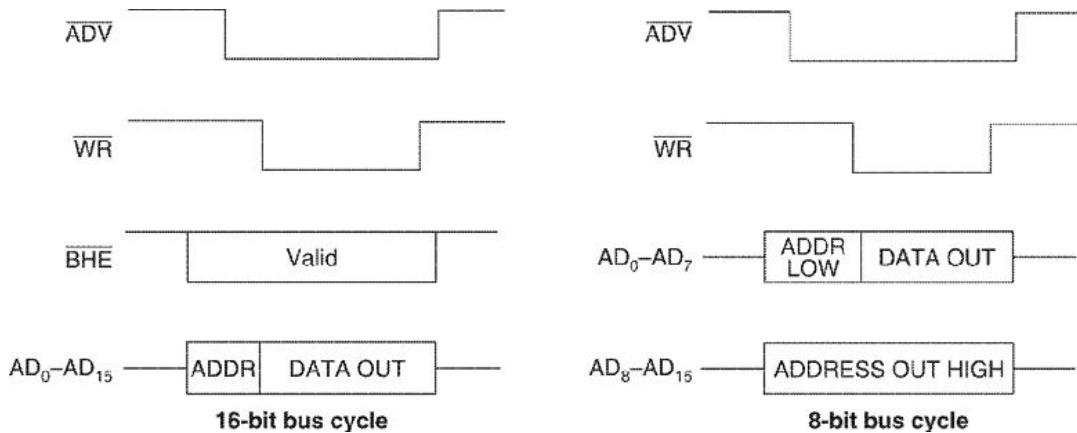


Figure 12.20 Address valid strobe mode.

The external memory interface circuit for address valid strobe mode (16-bit bus cycle) is shown in Figure 12.21. Only one even and one odd memory (2716/6116) bank have been interfaced.

In case $\text{CCR.2} = 0$ and $\text{CCR.3} = 0$, the interface becomes the simplest as $\overline{\text{WRH}}$ and $\overline{\text{WRL}}$ signals are provided along with $\overline{\text{ADV}}$. The mode is known as Write Strobe with Address Valid Strobe Mode. Figure 12.22 shows the signals in case of the 16-bit bus cycle ($\text{CCR.1} = 1$) and the 8-bit bus cycle ($\text{CCR.1} = 0$). The memory interface circuit is shown in Figure 12.23 for the 16-bit bus cycle.

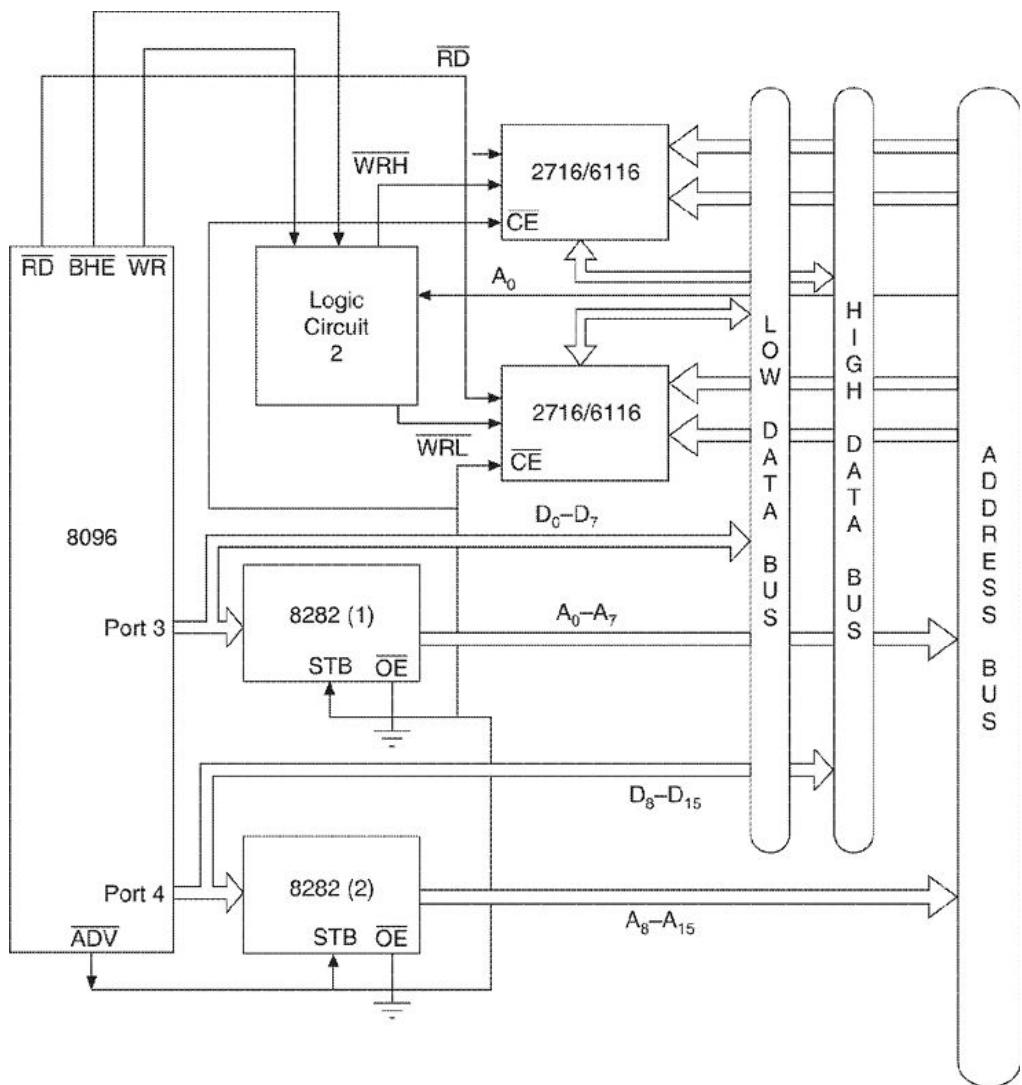


Figure 12.21 Memory interface (16-bit bus cycle) in address valid strobe mode.

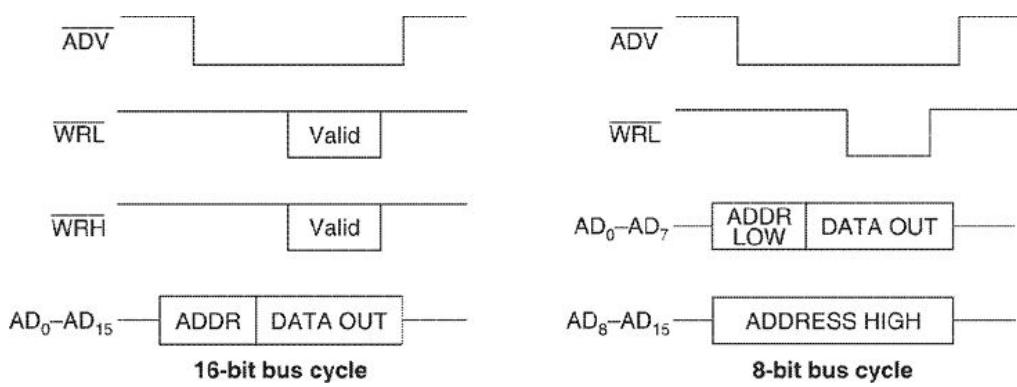


Figure 12.22 Write strobe with address valid strobe.

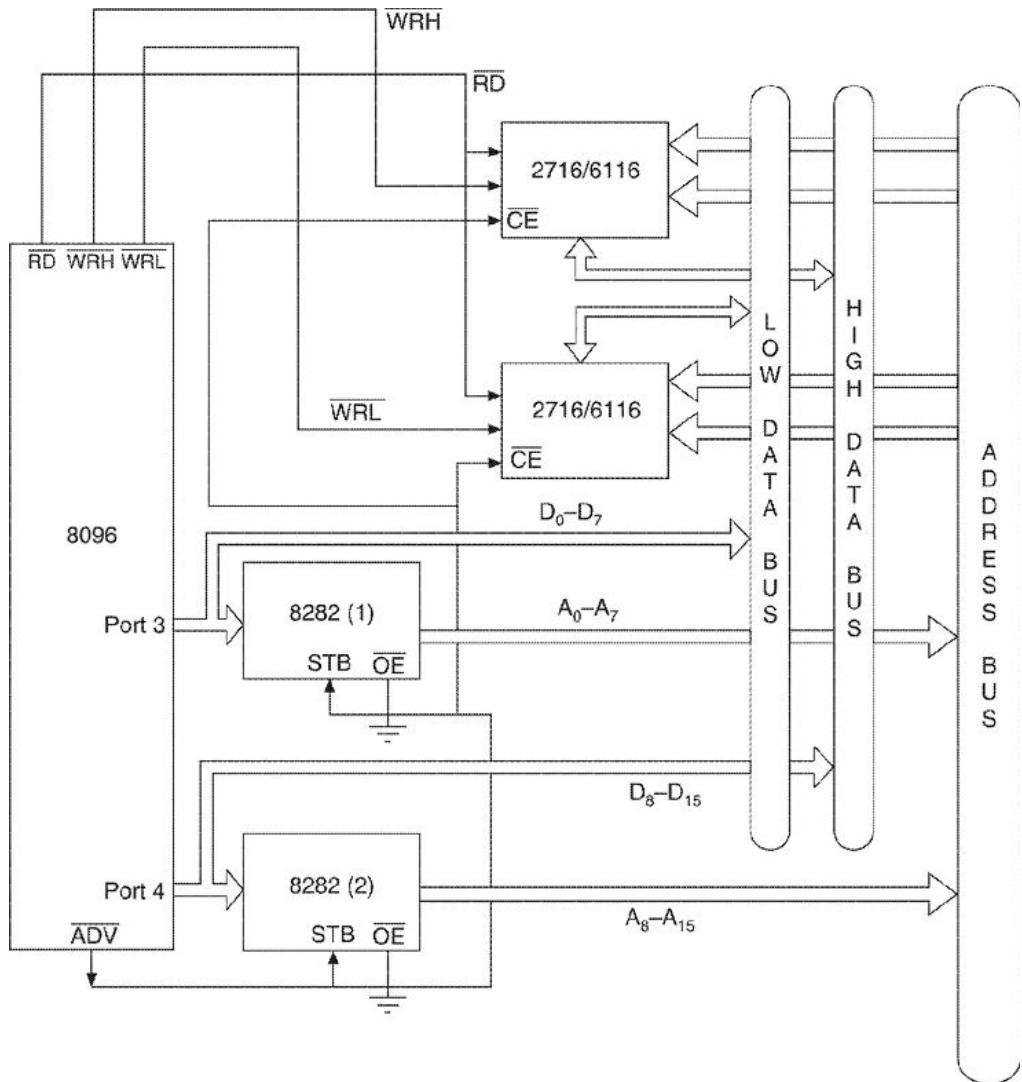


Figure 12.23 Memory interface (16-bit bus cycle) in write strobe with address valid strobe.

Wait state control: For slow memory interface, the wait states are inserted when the READY pin becomes 0. Bits 4 and 5 of the CCR program the internal ready control. When the READY pin is pulled low, the wait states will be inserted till either the READY pin becomes high or the number of wait states equal the number specified in CCR.4 and CCR.5, whichever comes first. The internal ready control vis-a-vis the IRC0 (CCR.4) bit and the IRC1 (CCR.5) bit is shown below.

<i>IRC1</i>	<i>IRC0</i>	Description
0	0	Limit to 1 wait state
0	1	Limit to 2 wait state
1	0	Limit to 3 wait state
1	1	Internal ready control disabled

Securing the data/code in memory: Often it is desired that a part of the memory must be secured from unintentional and intentional read/write operations. The memory may contain control constants, limits, initial settings or even program code. CCR.6 (LOC0) and CCR.7 (LOC1) enable the read/write protection of a portion of internal program memory from a program executing from the external memory.

Internal ROM/EPROM addresses 2020H to 3FFFH are protected from the reads and 2000H to 3FFFH are protected from the writes based on the values of LOC0 and LOC1 (CCR.6 and CCR.7) as per the table below.

<i>LOC1</i>	<i>LOC0</i>	<i>Protection</i>
0	0	Read and write protected
0	1	Read protected
1	0	Write protected
1	1	No protection

When read protection is in force then only the codes executing from internal memory can read the protected internal memory. However in case of write protection, the protected internal memory space cannot be written by the program executed from even internal memory.

12.8 I/O INTERFACING

The 8096 architecture provides five I/O ports which can be effectively used for the interfacing of any input–output device. It must however be understood that these ports have certain alternate functions as well, which may need to be sacrificed in case these are used.

Figure 12.24 shows a simple interface of 4 × 4 keyboard with the 8096 using one of the ports—let us say P1. The keyboard (refer to Section 7.5.1 for details) has been divided into rows and columns, and connected to port lines in the following manner.

Row 1 to Row 4 : Port lines 0 to 3

Column 1 to Column 4 : Port lines 4 to 7

A particular key, when pressed, can be identified by its row and column numbers. However, to find out whether a particular key has been pressed or not, the 8096 will have to scan the keys regularly. The algorithm for this will be simple.

For row no. $j = 1$ to 4

Send level 1(5 V) signal to row no. j

For column no. $k = 1$ to 4

 Read signal level at column no. k

 If level = 1, then go to keycode

 else repeat column

repeat row

Key code: Input code for key (j, k) .

Like keyboard, LED display is often used in applications requiring the 8096 as stand-alone system. These may differ from simple traffic light controller to complex process controller applications.

Single LEDs can be used for ON/OFF indication or alarm annunciation.

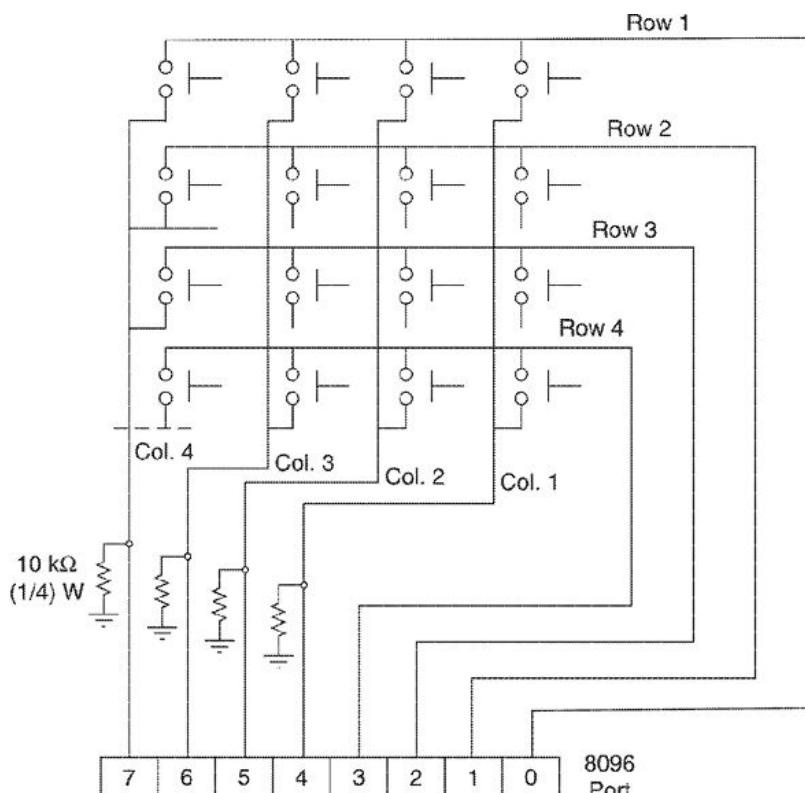


Figure 12.24 4 x 4 keyboard interface.

Figure 12.25 shows the interfacing of eight LEDs to the 8096 through a port in the common anode fashion. A level 0 at any port line will glow the LED connected to the line. Alternate 0 and 1 on the line with some wait period in between will enable the blinking of LED and this may be used for alarm indication.

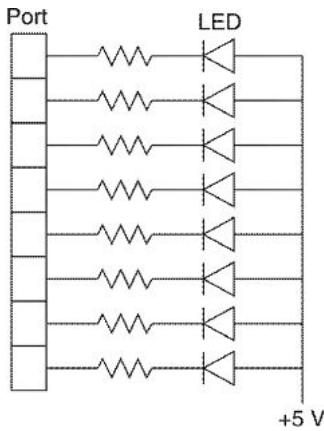
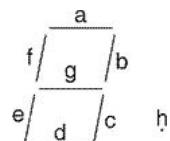


Figure 12.25 LED interface (common anode).

The seven-segment LED displays are very popular in instruments, industry as well as various appliances, and are used to display decimal digits. The seven-segment display looks as shown in the diagram below; a, b, c, d, e, f, g and h are called the segments of the seven-segment display.



In fact, now it is the eight-segment display as the eighth segment ‘h’ has been added to display the decimal point. The coding of the digit to be displayed is done as follows.

Bit no.	7	6	5	4	3	2	1	0
Segment	a	b	c	d	e	f	g	h

Whichever segment we want to glow, we store binary ‘0’ at the particular position of the segment (this is applicable in case of common anode type seven-segment display whereas in common cathode it is just the opposite).

Now suppose we want to display ‘A’. As per the standard segment nomenclature, the segments to glow would be a, b, c, e, f and g. If we are using common anode type we will store ‘0’ at these locations. So, the equivalent code would be 11H. The display format for various digits can be similarly worked out.

The serial interface of seven-segment LED display is shown in Figure 12.26.

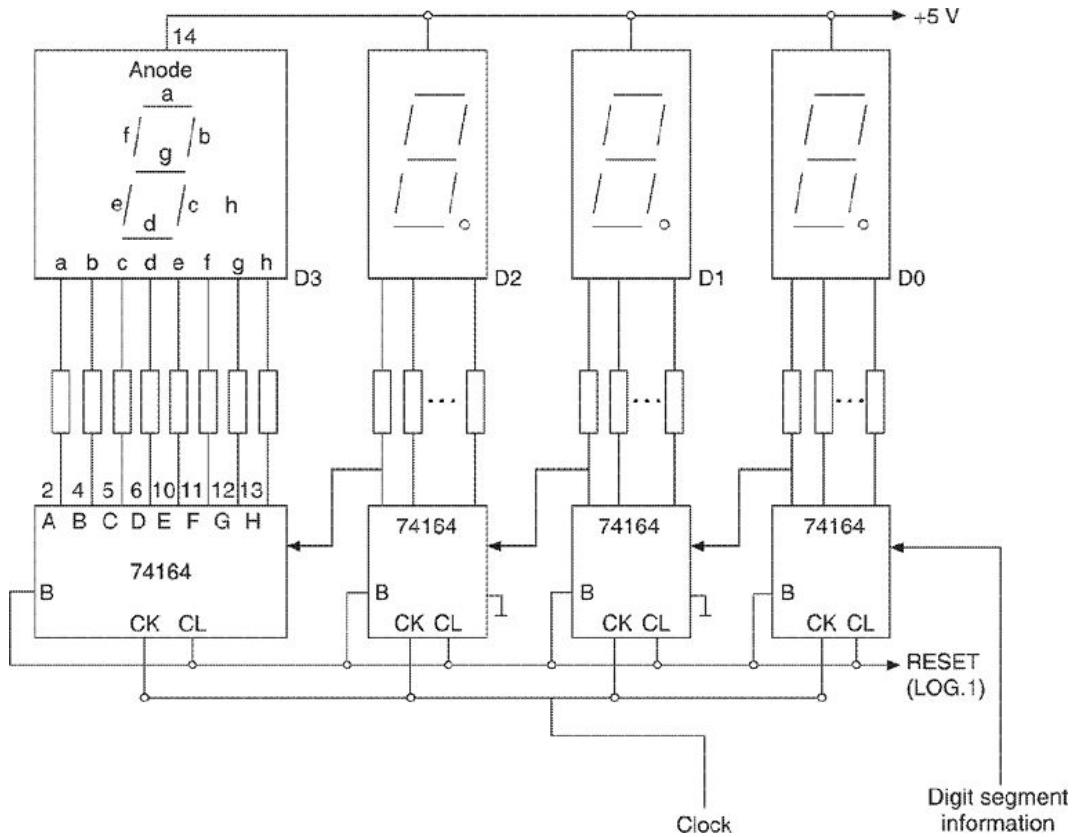


Figure 12.26 Serial interface of seven-segment LED display.

The interface circuit contains four modules of seven-segment display connected in serial mode through the 74164 shift register. The 8-bit display code for modules can be stored in four memory locations.

We input MSB of the digit at data input and give a clock. Then we input the next bit and again give a clock. So in total eight clock pulses, we can display a digit. For the next digit the same process is repeated and this way we can display as many number of digits as we want.

Two lines of any port can be used for inputting clock and digit segment informations. We determine the code of the digit to be displayed and store the codes of the digit in the consecutive memory locations. The code for D3 will be stored first, and then for D2, and so on.

It is necessary to have Digit and Segment Counters. The maximum count for a Segment Counter is 8 and for a Digit Counter, it is 4. After every bit/segment transfer we decrement the Segment Counter and after every digit the Digit Counter is decremented. As soon as both the counters become 0, the digit transfer is completed.

12.9 INTERRUPTS

The 8096 has eight sources of interrupts as shown in Table 12.4. In addition, there is a TRAP instruction which is a software generated

interrupt. However, this is not available to the user. The vector locations for the Interrupt Service Routines (ISRs) for these interrupts are stored in the predefined locations. Table 12.4 also shows the vector locations where the locations for different Interrupt Service Routines can be stored and priority assigned to these interrupts.

Table 12.4 Interrupt vector locations

<i>Interrupt source</i>	<i>Priority</i>	<i>Vector location</i>
Timer Overflow	0 (Lowest)	2000H
A/D Conversion Complete	1	2002H
HSI Data Available	2	2004H
HSO Execution	3	2006H
HSI.0	4	2008H
Software Timers	5	200AH
Serial I/O	6	200CH
External Interrupt	7 (Highest)	200EH

In order to get acknowledged, an interrupt should occur at least four state times before the end of instruction. All the interrupts are tested for 0 to 1 transition. There are two registers used for interrupt control.

1. Interrupt Pending Register
2. Interrupt Mask Register.

Whenever an interrupt request is detected, an entry is made in the Interrupt Pending Register. This register has one bit reserved for each interrupt source. Depending on the source of the interrupt, the corresponding bit is made 1 by the 8096 hardware. Though the hardware priority for the interrupts is defined, the user can dynamically allocate priorities to different interrupts as per the need by selectively enabling/disabling various interrupts using the Interrupt Mask Register. This register has a defined bit position for each interrupt. When a bit is made 0, the corresponding interrupt is disabled. It can be enabled by making the particular bit 1. The formats of the Interrupt Pending Register and the Interrupt Mask Register are same and shown in Figure 12.27.

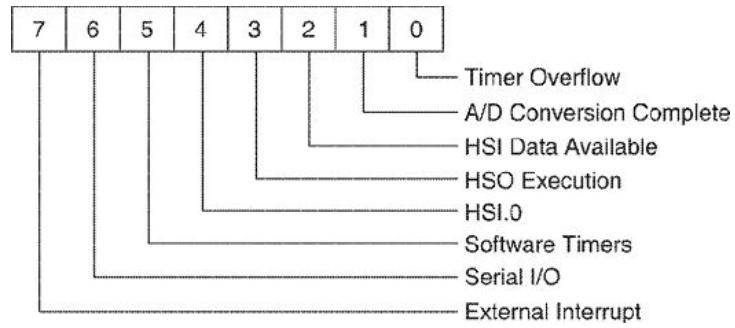


Figure 12.27 Interrupt mask and pending registers.

It must be noted that the Interrupt Mask Register in the 8096 is a misnomer and is different from the 8085. Here it is actually the Interrupt Enable Register (bit = 1 \square Interrupt enable, and bit = 0 \square Interrupt disable).

The block diagram of the 8096 interrupt system is shown in Figure 12.28. In addition to selectively enabling/disabling of interrupts, the whole interrupt system can be enabled or disabled by executing EI (Enable Interrupt) or DI (Disable Interrupt) instructions.

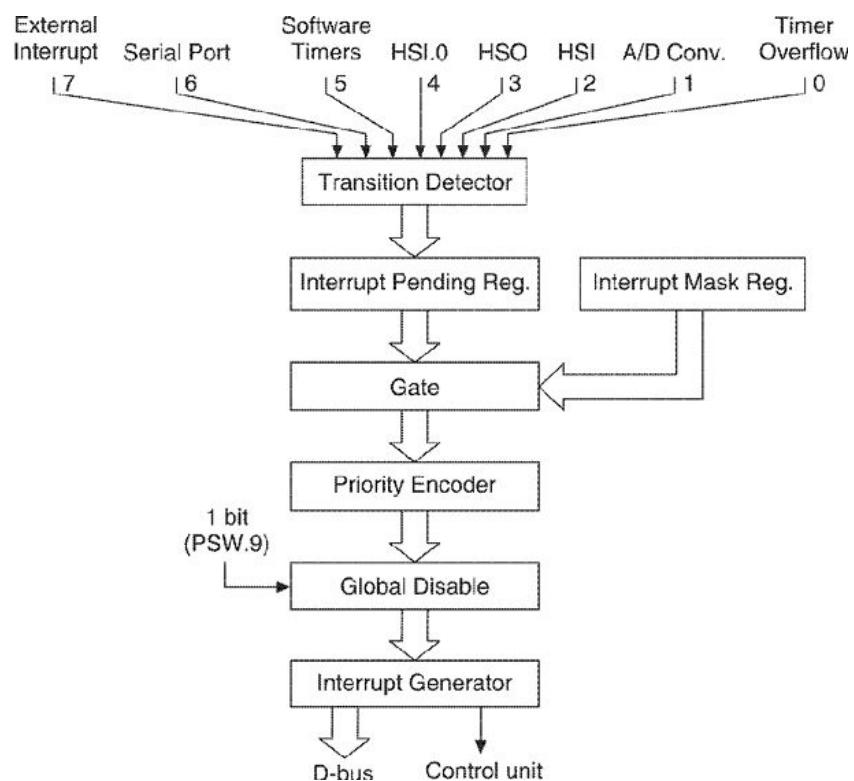


Figure 12.28 Block diagram of interrupt system.

Certain finer controls on the interrupt system sources are exercised by the I/O Control Register 1 and High Speed Output Command Register (Figure 12.29). These are discussed separately.

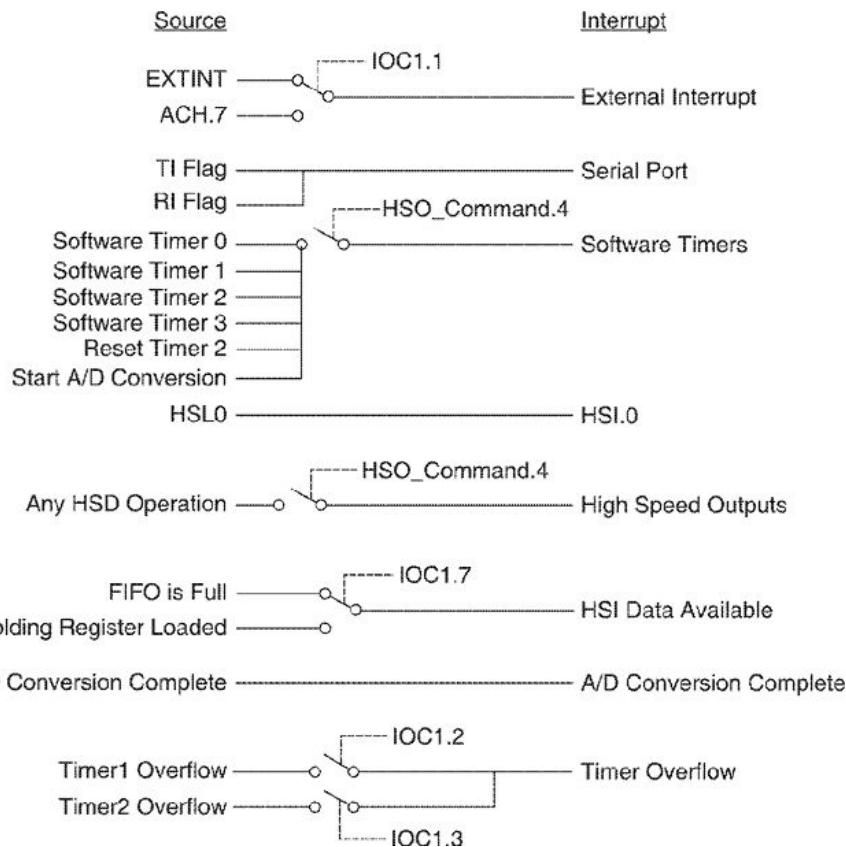


Figure 12.29 All possible interrupt sources.

12.10 INPUT/OUTPUT PORTS

The 8096 contains four 8-bit I/O ports which are shared with other functions as well. The ports are buffered both at input and output.

Port 0: It is an 8-bit high impedance, input only port. Its pins can be used as digital inputs and/or analog inputs to the on-chip A/D converter. These pins are also mode input to EPROM parts in the programming mode.

Port 1: It is a quasi-bidirectional port and can be used for sending/receiving 8-bit parallel data.

Port 2: It is a multifunction port since six of its pins are shared with other functions of the 8096 as shown in Table 12.5.

Table 12.5 Port 2 alternate functions

Port	Function	Alternate function	Controlled by
P2.0	output	TXD (Serial port transmit)	IOC1.5
P2.1	input	RXD (Serial port receive)	N/A
P2.2	input	EXTINT (External interrupt)	IOC1.1
P2.3	input	T2CLK (Timer 2 input)	IOC0.7

P2.4	input	T2RST (Timer 2 reset)	IOC0.3
P2.5	output	PWM (Pulse width modulation)	IOC1.0
P2.6	quasi-bidirectional		
P2.7	quasi-bidirectional		

Port 3 and Port 4: Port 3 and Port 4 can be used as bidirectional ports if the external memory is not required. In case of the external memory, these are used for the multiplexed address and data lines (P3.0–AD₀, P3.7–AD₇; P4.0–AD₈, P4.7–AD₁₅).

12.11 I/O CONTROL REGISTERS

There are two I/O control registers—IOC0 and IOC1.

IOC0

Located at 0015H, this is used to control Timer 2 and High Speed Input lines. Figure 12.30 shows the format of IOC0.

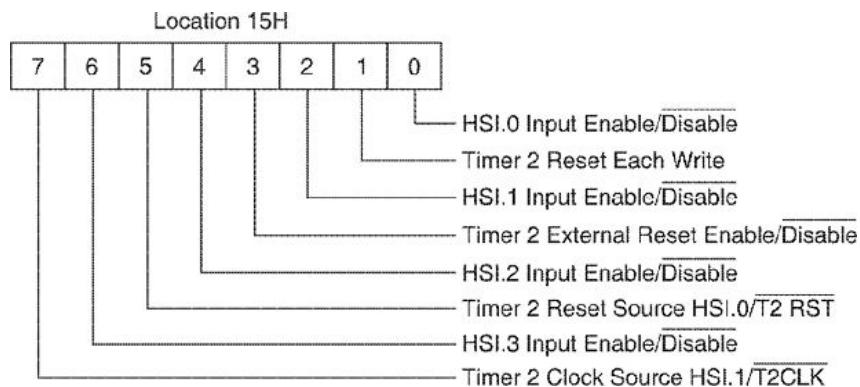


Figure 12.30 I/O control register 0 (IOC0).

IOC1

Located at 0016H, this is used to control some interrupt sources, PWM and High Speed Output lines 4 and 5. The format is shown in Figure 12.31.

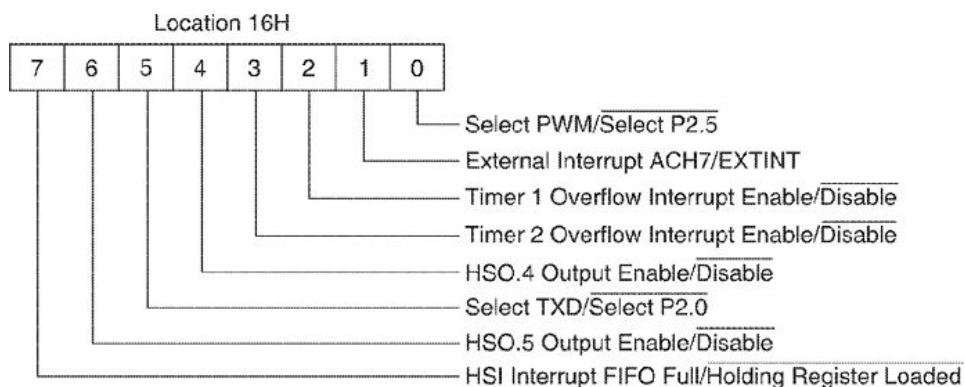


Figure 12.31 I/O control register 1 (IOC1).

12.12 I/O STATUS REGISTERS

There are two I/O status registers—IOS0 and IOS1

IOS0

Located at 0015H, it holds the current status of the HSO lines and CAM (Figure 12.32).

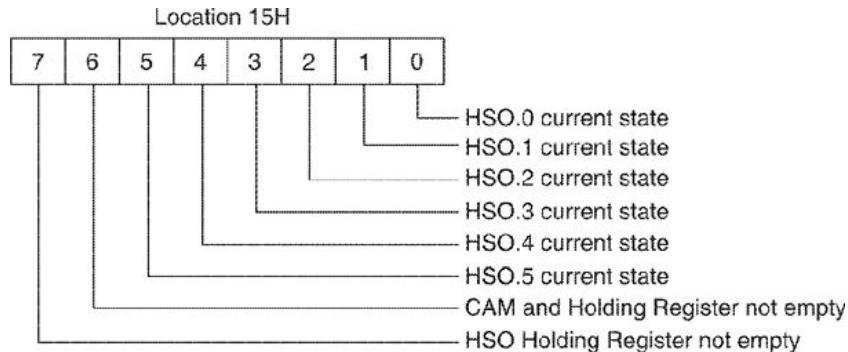


Figure 12.32 HSIO status register 0 (IOS0).

IOS1

Located at 0016H, it contains the status bits of Timers and High Speed Input unit (Figure 12.33).

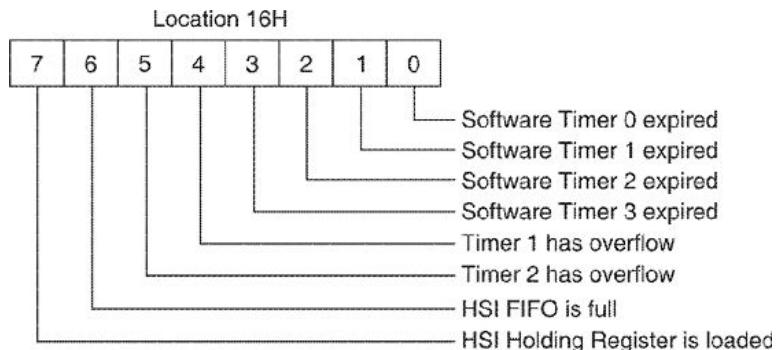


Figure 12.33 HSIO status register 1 (IOS1).

12.13 PROGRAMMING OF THE 8096 RESOURCES

The 8096 microcontroller provides the following on-chip resources.

- Timer/Counters
- High Speed Inputs
- High Speed Outputs
- Serial Input–Outputs

- Analog-to-Digital Converter
- Analog (Digital-to-Analog Converter) Output

We shall now discuss these resources and their programming to suit the requirements of the applications.

12.13.1 Timers

There are two hardware and four software timers provided in the 8096. These are 16-bit timers and used for various functions. The hardware timers are called Timer 1 and Timer 2 and are described in this section. The software timers are described along with the High Speed Output facility.

Timer 1

Timer 1 is used to provide real-time clock for external events which are recorded on High Speed Input lines (HSI) or which are generated on High Speed Output lines (HSO) of the 8096. The input clock is (XTAL1 frequency/24). In other words, it is clocked once every eight state times. Since for 12 MHz input clock the state time is 0.25 ms, the minimum time period of Timer 1 clock will thus be 2 ms. Timer 1 can be reset only by executing reset (through RST instruction or by sending a pulse to the reset pin of the 8096).

Timer 2

The Timer 2 can have one of the following two sources as clock.

- Timer 2 clock (Port 2.3–Pin 44/34)
- High Speed Input line no. 1 (Pin 25/53/4)

The selection of the clock source can be done by the user by programming the bit 7 of I/O Control Register 0 (bit 7 = 1 – HSI.1, bit 7 = 0 – T2CLK).

Since Timer 2 can be used for generating High Speed Outputs, the maximum speed of Timer 2 is once per eight state times. Timer 2 is incremented by transitions (one count by each rising or falling edge). Timer 2 can be reset

- by executing reset, or
- by setting bit 1 of I/O Control Register 0 = 1, or
- by triggering HSO channel 14 (0EH), or

- by making bit 3 of I/O Control Register 0 = 1 and then by making T2RST (Port 2.4, Pin 42/36) or High Speed Input 0 (Pin 24/54/3) = 1.

It gives ample scope for the manipulation of Timer 2 both externally as well as internally. Figure 12.34 shows the logical diagram to explain the way Timer 2 can be manipulated.

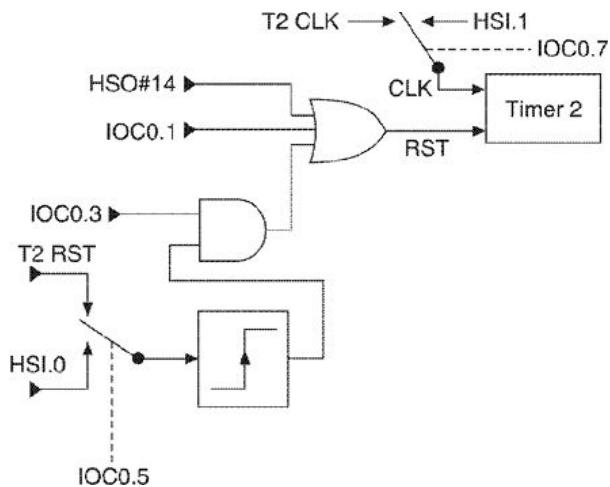


Figure 12.34 Timer 2 clock and reset options.

Timer Interrupts

The timer overflow is one of the eight sources of interrupts in the 8096. Both Timer 1 and Timer 2 can be used to trigger a timer overflow interrupt. As soon as a timer overflow occurs, the flag bit 4 (for Timer 1) and 5 (for Timer 2) are set to 1 in I/O Status Register 1 (IOSI). The timer interrupts are controlled by setting bits 2 and 3 of I/O Control Register 1 to 1 as follows:

- IOC1 bit 2 = 1 Timer 1 overflow interrupt enable.
- IOC1 bit 3 = 1 Timer 2 overflow interrupt enable.

The enabled timer interrupt can go for interrupting the 8096. However, general enabling/disabling of the timer interrupts is controlled by the Interrupt mask register bit 0.

EXAMPLE 12.1

Design an 8096-based system for speed measurement in a car.

Solution:

The speed calculation of the car will require the use of timer/counters. Let us assume that the existing speedometer cable makes 1 revolution for every 1 metre travel. This assumption is normally true in many cars. The

number of revolutions in 1 km will thus be 1000. We will, therefore need two timer/counters for *speed calculation*.

Timer 1—For time measurement.

Timer 2—To count rotations of the speedometer cable.

For XTAL1 input of 12 MHz at Timer 1, the clock period = 0.25 ms

Thus, the increment in count will occur after every $8 \times 0.25 = 2$ ms

Timer overflow interrupt for Timer 1 will occur after 2^{16} counts, i.e. after $(2^{16} \times 2)$ ms = 0.13 second.

The speedometer cable can be connected to a shaft encoder which will generate a TTL level pulse after every revolution. The output of the shaft encoder can be connected to the T2CLK input of Timer 2.

We must remember that Timer 2 is incremented by both transitions, i.e. 0 to 1 and 1 to 0 transitions of clock. Thus the count value in Timer 2 will be twice the number of revolutions of speedometer cable.

Now, let us work out the various timings considering 500 km/h as the maximum speed and 1 km/h as the minimum speed.

For 500 km/h speed:

No. of revolutions of the speedometer cable in 1 hour (60 \times 60 seconds) = 500 \times 1000

No. of revolutions of the speedometer cable in 1 seconds = 138.88

No. of shaft encoder pulses in 0.13 second (i.e. between two Timer 1 Overflow interrupts) = 18

Timer 2 count value = 36

For 1 km/h speed:

No. of revolutions of the speedometer cable in 1 second = $1000/(60 \times 60)$ = 0.277

No. of shaft encoder pulses in 0.13 second = $0.277 \times 0.13 = 0.036$

Thus, no pulse will occur whereas minimum 1 pulse is required to determine the speed.

Timer 2 count value = 0

Time for 1 revolution of the speedometer cable = $(60 \times 60)/1000$ = 3.6 seconds

It is clear that if we wish to determine speed from 1 km/h onwards, we must read Timer 2 only after 3.6 seconds. But the Timer 1 overflow interrupt will occur every 0.13 second.

The solution to this can be that ISR for the timer overflow interrupt should have a software counter which will count $3.6/0.13 = 28$ occurrences of the timer interrupt before reading Timer 2. For maximum

speed of 500 km/h, the Timer 2 count value will be $36 \square 28 = 1008$. For different speeds, the Timer 2 count value may be calculated and stored in memory in the form of a look-up table.

Let us work out the settings of different SFRs for Timer 1 and Timer 2.

Timer 1:

Timer overflow interrupt must be enabled by $\text{IOC1.2} = 1$

The interrupt will occur only if the timer overflow interrupt system is enabled in Interrupt Mask Register by Interrupt Mask Register (location 08H) bit 0 = 1.

Timer 2:

Timer 2 clock source is T2CLK (i.e. P2.3). It is set by $\text{IOC0.7} = 0$

Timer 2 overflow interrupt must be disabled by $\text{IOC1.3} = 0$

Timer 2 must be reset after every read by $\text{IOC0.1} = 1$ as part of the software after read instruction.

Other reset options must be disabled by (External reset disable) $\text{IOC0.3} = 0$

Finally:

$\text{IOC0} = 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 = 02H$

$\text{IOC1} = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0 = 04H$

Interrupt Mask Register = $0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 = 01H$

Now, software logic

Main Program:

- Load Interrupt Mask Register (location 08H) with 01H
- Load IOC0 (location 15H) with 02H
- Load IOC1 (location 16H) with 04H
- COUNT $\square 0$ (COUNT may be represented in one of the registers)

ISR for Timer overflow:

- Read IOS1 (location 16H)
- If $\text{IOS1.4} = 1$ then Go To ISR Timer 1
- Else error

ISR Timer 1

```

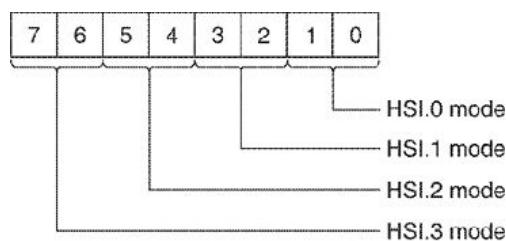
COUNT = COUNT + 1
If COUNT < 28
    Then Enable Interrupt
    Return
Else Read Timer 2
    Read speed from memory look-up table corresponding to
    value in Timer 2.
    Display speed
    Reset Timer 2 by IOC0.1 = 1 ( i.e. load IOC0 with 02H)
    COUNT = 0
    Enable Interrupt
    Return

```

12.13.2 High Speed Inputs

In real-time control applications, it is often required to know and record the various signal events along with their times of occurrences. The 8096 provides four High Speed Input (HSI) lines, on which a total of eight events can be recorded. HSI.2 and HSI.3 share pins with High Speed Output pins 4 and 5. The function of these pins (HSI.2/HSO.4—Pin 26/52/5 and HSI.3/HSO.5—Pin 27/51/6) is controlled by I/O Control Register 0 (bits 4 and 6) and I/O Control Register 1 (bits 4 and 6).

There are four possible modes of each High Speed Input line. The modes basically signify the type of signal events that will be recorded. Figure 12.35 shows the HSI_MODE register along with the event types. For setting up modes for different HSI lines, the mode register (SFR at location 03H) should be appropriately programmed. In mode 0, which takes eight positive transitions as an event, the input speed can be one transition per state time. For other modes the input speed is one transition per eight state times.



where each 2-bit mode control field
defines one of the four possible modes:

- 00 Eight positive transitions
- 01 Each positive transition
- 10 Each negative transition
- 11 Every transition
(positive and negative)

Figure 12.35 The HSI_MODE register.

The block diagram giving the internal architecture of High Speed Input unit is given in Figure 12.36. Depending on the modes set, the change detector circuit generates event (triggered input) which is stored in FIFO along with the Timer 1 contents. The FIFO is 20-bit wide with 4 bits meant for status of four HSI lines and 16 bits for Timer 1 contents. Seven entries (seven events) in total can be stored in FIFO. The holding register is used to store one additional event when FIFO is full.

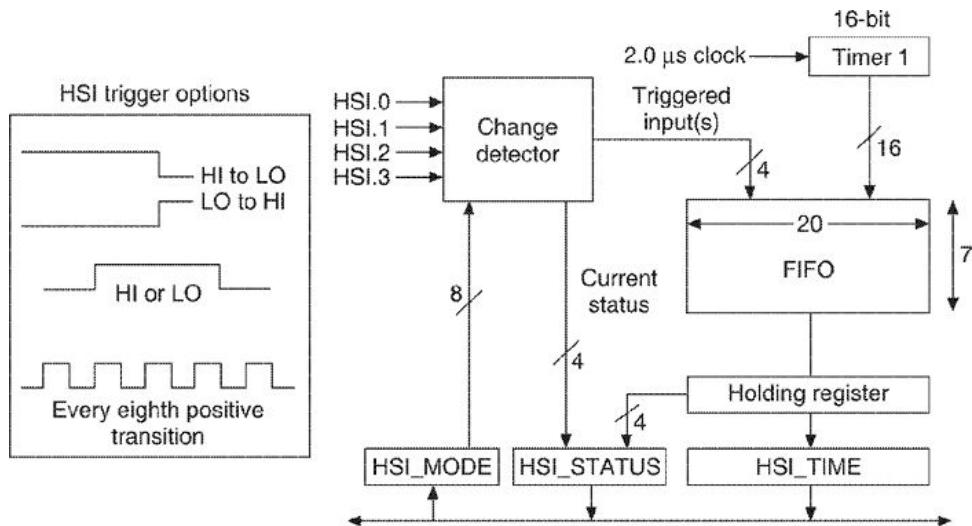


Figure 12.36 High speed input unit.

The HSI_STATUS register format is shown in Figure 12.37. The register shows the current status of HSI pins. Two bits of the register are dedicated to each HSI pin. The lower bit indicates whether or not the event has occurred on the pin at the time given, i.e. HSI_TIME. The upper bit indicates the current status of the pin.

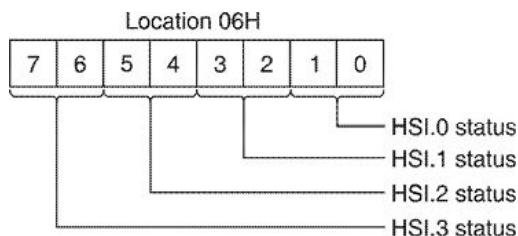


Figure 12.37 The HSI_STATUS register.

The FIFO can be read via the HSI_STATUS register and HSI_TIME register. These are special function registers at locations 06H and 04–05H. In addition, the status of FIFO is indicated in I/O status register 1 (bits 6 and 7).

If bit 6 = 1, FIFO is full. It contains at least seven entries.

If bit 7 = 1, Holding register is loaded, i.e. FIFO contains at least one entry.

High Speed Input unit can interrupt the 8096 through two interrupt sources, i.e. HSI.0 and HSI Data Available, as discussed in the section on interrupts.

Any transition on the HSI line 0 is taken for interrupt. Since HSI.0 can be used to trigger Timer 2 too, this may be very useful in applications where time-sequenced events are to be monitored/controlled. HSI Data Available interrupt can be so controlled that it will be generated every time a holding register is loaded (7th bit of the I/O Control Register 1 = 0) or it will be generated when FIFO contains minimum six entries (7th bit of the I/O Control Register 1 = 1).

12.13.3 High Speed Outputs

The 8096 can be programmed to trigger different events at pre-specified time. The events are:

- Starting of A/D conversion
- Resetting of Timer 2
- Generation of two interrupts
- Setting software timers
- Switching up to six output lines.

The six output lines of High Speed Output (HSO) unit are designated HSO.0–HSO.5. As previously stated, HSO.4 and HSO.5 are shared with HSI.2 and HSI.3 and this function is controlled by I/O Control Register 1.

The HSO unit contains (Figure 12.38) a Content Addressable Memory (CAM) in which the time of the event (16 bits) and HSO command (7 bits) to trigger the event are stored. These are stored through HSO_COMMAND and HSO_TIME registers which are special function registers. A total of eight entries can be stored in CAM and when it is full, then one entry can be stored in the Holding register. The control logic through MUX and Comparator checks the time data of various entries with the current time in Timer 1 or Timer 2. The HSO command specifies that the reference is from Timer 1 or from Timer 2. The output signal as mentioned in the HSO command is generated. The HSO command format is shown in Figure 12.39.

It specifies that:

1. Either one of the six HSO lines or the combination of two (HSO.0, HSO.1) and (HSO.2, HSO.3) can be set (high level) or cleared (low level). These events can be programmed to cause interrupt as well.
2. Any one of the four software timers can be programmed to generate interrupts. I/O Status Register 1 will specify which counter caused the interrupt.
3. Timer 2 or Timer 1 can be programmed to be reset and to cause interrupts.
4. A/D conversion can be triggered.

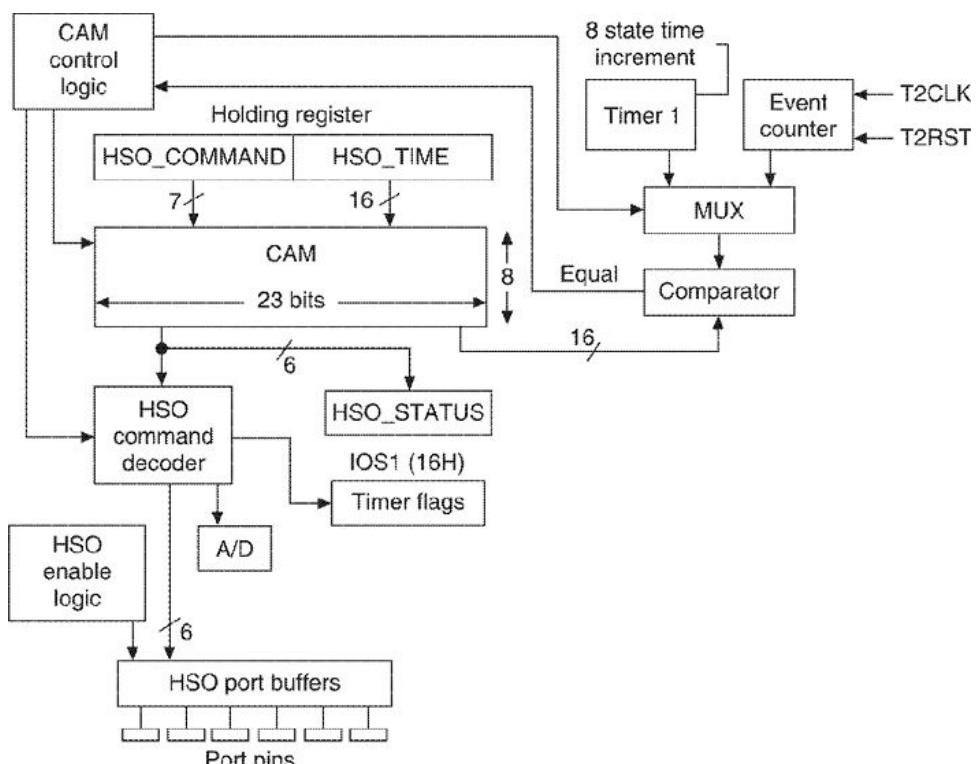


Figure 12.38 High speed output unit.

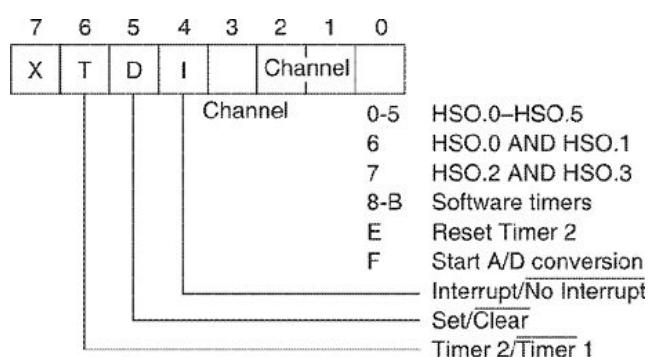


Figure 12.39 HSO command tag format.

All these events are triggered at pre-specified time. HSO command also specifies (bit 6) whether Timer 1 or Timer 2 (Event Counter) should be taken for time comparison.

It takes one state time to check one location of CAM. Thus, eight state times are required to check all the locations in CAM. Hence, the minimum time difference between two events will be eight state times. This is the reason why Timer 1 is incremented only once every eight state times. Timer 2 in HSO has also been synchronized to change once per eight state times. It should be noted that bit 5 is ignored for command channel 8 to 0FH.

While writing into CAM, it is necessary to know about the status of the CAM and the holding register. The I/O status register 0 specifies this through bits 6 and 7. If

Bit 7 = 0: Holding register is empty.

Bit 6 = 0: Holding register and at least one entry in CAM is empty.

Depending on the status, further entries should be stored in CAM.

We shall present two examples—one concerning HSI-HSO and the other for external interrupt and HSO.

EXAMPLE 12.2

Four switches (SW1 to SW4), when manually/automatically put on, initiate the actions on the machine which take place after a fixed time.

SW1 initiates Relay1 after 20 seconds.

SW2 initiates Relay4 after 5 seconds.

SW3 initiates Relay2 after 30 seconds.

SW4 initiates the action of checking after 30 seconds the status of Relay1 and Relay4.

If both Relays 1 and 4 are ON, then (a) Realy3 is switched on and (b) Relay1 and Relay4 are switched off.

These relays will initiate some machine actions and further actions will be scheduled through programming. The task before us is to use the 8096 to sense the 0 to 1 rise in four switch lines SW1 to SW4 and switch on different relays after a delay of few seconds as specified.

Solution:

The four switch lines can be connected to four HSI lines—HSI.0 to HSI.3—and 0 to 1 transition can be sensed and stored in FIFO of HSI unit. The HSI unit works with Timer 1. Thus, the contents of Timer 1 and the status of four lines will be stored whenever there is a change.

If we consider that the clock frequency of the 8096 is 12 MHz, then the time period for Timer 1 will be 2 ms. The 16-bit Timer 1 register will overflow in 2×2^{16} ms = 0.13 second. Thus, any action with a delay of a few seconds cannot be scheduled using Timer 1 on the HSO lines. Another option is to program the HSO unit to work on the external clock at T2CLK. This would require a hardware clock generator which can output 1 second clock pulse. This can be connected at T2CLK.

Figure 12.40 explains the complete scheme. The input in HSI Holding register will cause an interrupt. The ISR should determine the line (HSI.0 to HSI.3) where the transition has caused the interrupt. The corresponding future actions can then be set in HSO lines (HSO.0 to HSO.3) as specified in the application. HSO.4 and HSO.5 lines cannot be used since they are used as HSI.2 and HSI.3.

When 0 to 1 transition in SW4 line occurs, the status of Relay1 and Relay4 needs to be checked after 30 seconds and if both are ON, then Relay3 is to be switched on. This task cannot be achieved by directly connecting Relay3 to HSO.3. Therefore, HSO.3 has been used to generate a software timer interrupt. The ISR should check the status of Relay1 and Relay4. Relay3, therefore, is connected through one of the port lines, say P1.3. Also the status of different relays needs to be read. The status outputs of relays R1 to R4 are connected to port lines as shown in Figure 12.40.

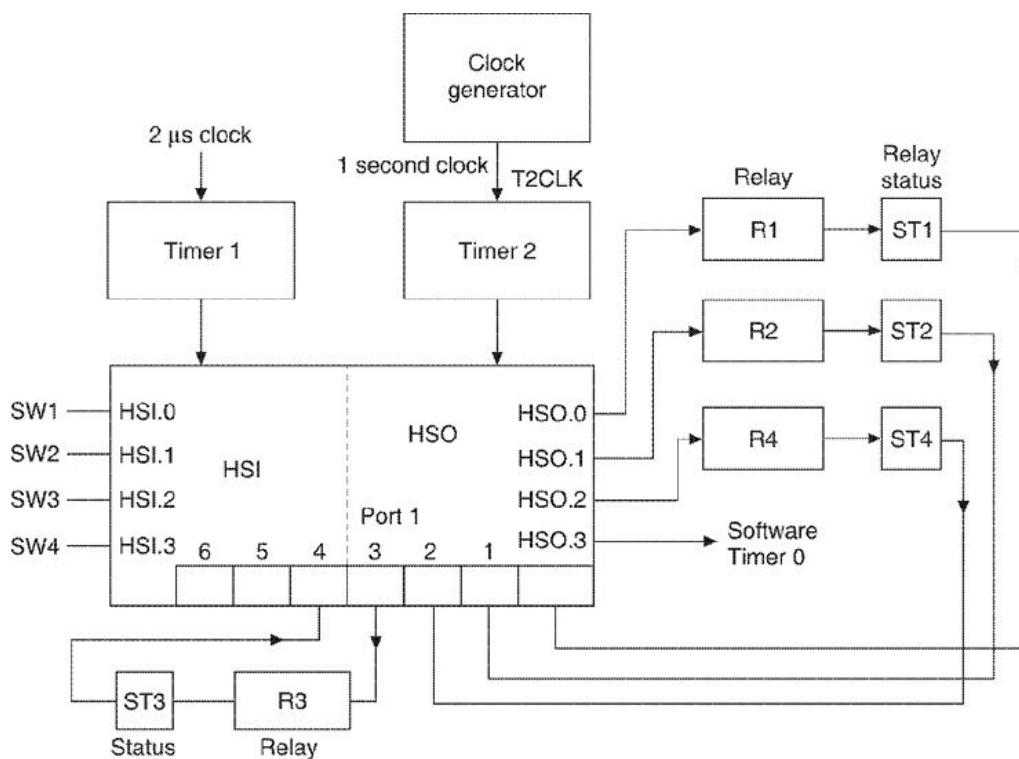


Figure 12.40 HSI-HSO example.

Let us now set the different SFRs of the 8096.

- (a) HSI.2/HSO.4 and HSI.3/HSO.5 lines are used as HSI.2 and HSI.3 lines. This is programmed by
 - enabling HSI.0 to HSI.3 through IOC0 (IOC0.0 = 1, IOC0.2 = 1, IOC0.4 = 1, IOC0.6 = 1).
 - disabling HSO.4 and HSO.5 through IOC1 (IOC1.4 = 0, IOC1.6 = 0).
- (b) Timer overflow interrupt must be disabled by
 - disabling Timer 1 and Timer 2 overflow interrupts through IOC1 (IOC1.2 = 0, IOC1.3 = 0).
 - disabling Timer overflow interrupt in Interrupt Mask Register (Interrupt Mask Register bit 0 = 0).
- (c) HSI Data Available interrupt must be enabled in Interrupt Mask Register (Interrupt Mask Register bit 2 = 1).
- (d) HSI interrupt must be selected as Holding Register Loaded (IOC1.7 = 0).
- (e) HSI mode for all four HSI lines should be set as “each positive transition”, i.e. mode 01 (HSI Mode Register = 0 1 0 1 0 1 0 1 = 55H).
- (f) Timer 2 must be selected for HSO (HSO Command Register bit 6 = 1) every time HSO command is sent.
- (g) Timer 2 clock source must be selected as T2CLK (IOC0.7 = 0).
- (h) Software Timer interrupt must be enabled by making Interrupt Mask Register bit 5 = 1.
- (i) Timer 2 must be reset (IOC0.1 = 0) whenever required. External reset must also be disabled (IOC0.3 = 0).

Finally:

IOC0	= 0 1 0 1 0 1 0 1 = 55H
IOC1	= 0 0 0 0 0 0 0 = 00H
HSI Mode Register	= 0 1 0 1 0 1 0 1 = 55H
Interrupt Mask Register	= 0 0 1 0 0 1 0 0 = 24H (Only HSI Data and SW Timer interrupts are enabled)

Now, software logic

Main Program:

- Load IOC0 by 55H
- Load IOC1 by 00H
- Load HSI Mode Register by 55H

- Load Interrupt Mask Register by 24H
-
-

ISR-HSI data available

- Read HSI Status Register
- Read Timer 2 SFR (location 0CH and 0DH)
- Determine the event source, i.e. HSI.0, HSI.1, HSI.2 or HSI.3 by checking different bits of HSI Status register (bit 0 for HSI.0, bit 2 for HSI.1, bit 4 for HSI.2 and bit 6 for HSI.3)
- Determine HSO command and HSO Time for each HSI event.
- Load HSO command and HSO Time

Since Timer 2 is incremented on both rising edge and falling edge, HSO TIME entered should be doubled to account for the desired delay. It must also be noted that bit 5 of HSO Command is ineffective for channels 8 to F.

- For HSI.0 event

HSO TIME = Timer 2 contents + 40 seconds = Timer 2 contents + 0028H

HSO COMMAND = x 1 1 0 0 0 0 0 = 60H

Load HSO TIME

Load HSO COMMAND

Return

- For HSI.1 event

HSO TIME = Timer 2 contents + 10 seconds = Timer 2 contents + 000AH

HSO COMMAND = x 1 1 0 0 0 1 0 = 62H

Load HSO TIME

Load HSO COMMAND

Return

- For HSI.2 event

HSO TIME = Timer 2 contents + 60 seconds = Timer 2 contents + 003CH

HSO COMMAND = x 1 1 0 0 0 0 1 = 61H

Load HSO TIME

Load HSO COMMAND

Return

- For HSI.3 event

HSO TIME = Timer 2 contents + 60 seconds = Timer 2 contents +
003CH

HSO COMMAND = x 1 0 1 1 0 0 0 = 58H

Load HSO TIME

Load HSO COMMAND

Return

ISR for Software Timer

Check bit 0 of IOS1 register to reconfirm Software Timer 0 interrupt.
If bit 0 = 0, then Error Return

Else, if (P1.0 AND P1.2) = 1

then P1.3 = 1

Load HSO COMMAND = x 1 0 0 0 0 0 0 = 40H and HSO TIME
= Timer 2 to switch off Relay1

Load HSO COMMAND = x 1 0 0 0 0 1 0 = 42H and HSO TIME
= Timer 2 to switch off Relay4

Return

Else Return

EXAMPLE 12.3

In a chemical factory, the gas detector continuously checks the leakage of any poisonous gas and on detecting any leakage, water spray is immediately started to save the workers from harmful effects of the gas. The spray is stopped after five minutes and the gas leakage is rechecked (while repairing). If gas leakage persists, then spray is started again for five minutes and the operation repeats. Design an 8096-based system to perform these functions.

Solution:

The output of the gas detector is connected to the EXTINT (external interrupt input). The spray is controlled through a relay connected to one of the port pins, say P1.0. By writing a 1 at P1.0, the relay will be switched on to start the spray.

The five minute interval can be measured through Timer 2, using a hardware clock generator which can output 1 second clock pulse. This can be connected at T2CLK. Figure 12.41 explains the complete scheme. The HSO.0 line can be connected to switch off the relay.

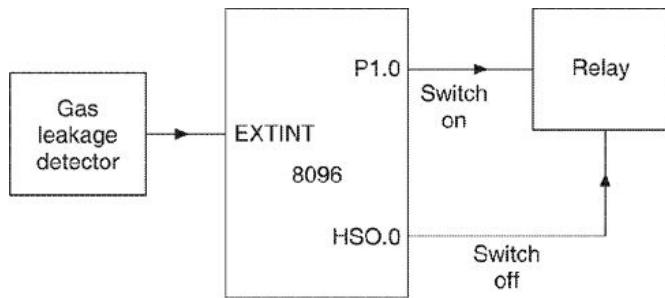


Figure 12.41 Interrupt example.

The sequence of the operation will be somewhat like the one mentioned below.

- External Interrupt is selected and the interrupt through ACH7 is disabled by making IOC1 register bit 1 = 0.
- External Interrupt is enabled at the Interrupt Mask Register. Other interrupts are masked.
- Timer 2 clock must be selected by bit 6 of HSO Command Tag Register = 1, every time the HSO Command is sent.
- Timer 2 clock source must be selected as T2CLK by IOC0.7 = 0.
- Timer 2 must be reset by IOC0.1 = 1, whenever required. Thus, the external reset must also be disabled by IOC0.3 = 0.
- Thus, finally:

$$\text{IOC0} = 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 = 02H$$

$$\text{IOC1} = 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 = 00H$$

$$\text{Interrupt Mask Register} = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 = 80H$$

- Now, software logic

Main Program:

Load IOC0 by 02H

Load IOC1 by 00H

Load Interrupt Mask Register by 80H

—

—

—

ISR EXTINT

Load P1 by 01H to make P1.0 = 1.

Load IOC0 by 02H to reset Timer 2

HSO Time = 600 = 0258H

HSO Command = x 1 1 0 0 0 0 0 = 60H

It is assumed that 0 to 1 transition at HSO.0 will switch off the relay. Since Timer 2 is incremented on both rising edge and falling edge, the HSO time entered should be doubled to account for the desired delay.

Load HSO Time

Load HSO Command

Enable Interrupt

Return

12.13.4 Serial Input/Output

The 8096 provides a serial port with two pins—TXD (for transmit) and RXD (for receive). These are shared with Port 2 pins P2.0 and P2.1. The selection is done by bit 5 of the I/O Control Register 1. If bit = 1, then the pin TXD is selected instead of Port 2.0.

The serial port is full duplex and receive buffered. Thus, transmission and reception can be simultaneous and the reception of the second byte can begin before the previously received byte has been read from the receive buffer. The following Special Function Registers are associated with serial I/O.

- Serial Transmit Buffer Register SBUF (TX)
- Serial Receive Buffer Register SBUF (RX)
- Serial Port Control Register SP_CON and Serial Port Status Register SP_STAT
- Baud Rate Register BAUD_RATE

The serial port has three asynchronous modes and one synchronous mode. The asynchronous modes are full duplex, i.e. they can transmit and receive at the same time.

Mode 0

Mode 0 is a synchronous mode which is used for shift register based serial I/O expansion. Serial data enters and exits from RXD. The 8096 sends eight shift pulses to the external shift register through TXD. The shift register can either send 8-bit data to the 8096 or receive the 8-bit data from the 8096 (Figure 12.42) with LSB as the first bit.

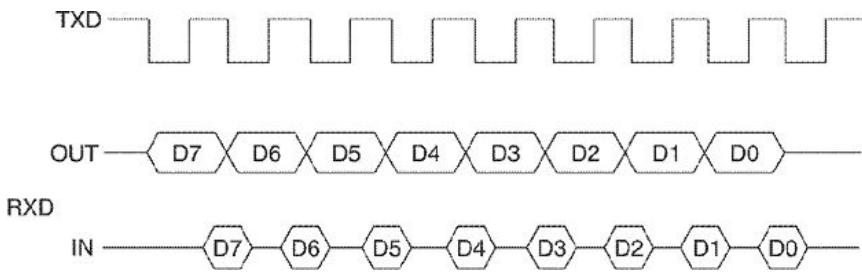


Figure 12.42 Serial port mode 0 timing.

Mode 1

Mode 1 is the standard asynchronous communication mode. 10-bit frames are received (RXD) or transmitted (TXD) (Figure 12.43). It is useful in interfacing the CRT terminal to the 8096. The TI and RI interrupt flags are set to indicate when the operations are complete. The TI is set when the last bit of the message has been sent, not when the stop bit is sent. Similarly, RI is set when the 8 data bits are received, not when the stop bit is received. The reading of the serial port status register will clear both TI and RI flags.

The transmit and receive functions are controlled by separate shift clocks. The transmit shift clock starts when the baud rate generator is initialized. The receive shift clock is reset when 1 to 0 transition happens, i.e when the start bit is received.

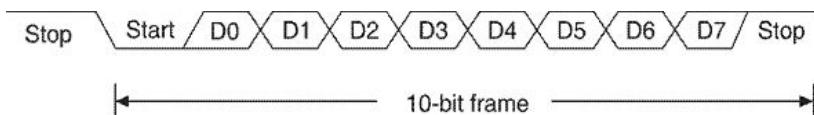


Figure 12.43 Serial port frame—mode 1.

Mode 2

Mode 2 is the asynchronous 9th bit recognition mode. 11-bit frames as shown in Figure 12.44 are transmitted (TXD) and received (RXD). The 9th bit is programmable and is used in multiprocessor communications. During transmission, the 9th bit can be assigned a 0 or a 1. On reception, if the 9th bit = 1, the serial port interrupt is activated.

Mode 3

Mode 3 is the asynchronous 9th bit mode. The data frame is same as that of mode 2. 11-bit frames are transmitted (TXD) and received (RXD) (Figure 12.44). On transmit, the 9th bit can be assigned a 0 or a 1. On reception, the 9th bit is stored and the serial port interrupt is activated regardless of its value. It is used in multiprocessor communications in conjunction with mode 2.

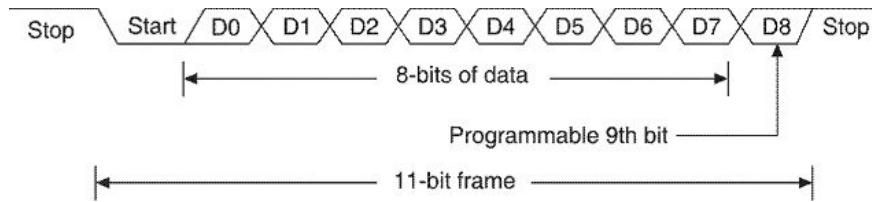


Figure 12.44 Serial port frame—modes 2 and 3.

Serial port control

Two special function registers SP_CON and SP_STAT, combined in one, are used for serial ports. Writing to location 11H accesses SP_CON, whereas reading location 11H accesses SP_STAT.

The register format is shown in Figure 12.45. The register is used to control the serial port resources. It is also used to read the status of these resources. TB8 is cleared after each transmission and both RI and TI flags are cleared when SP_STAT(not SP_CON) is accessed. In mode 0, the REN is made 0 and 1 alternately.

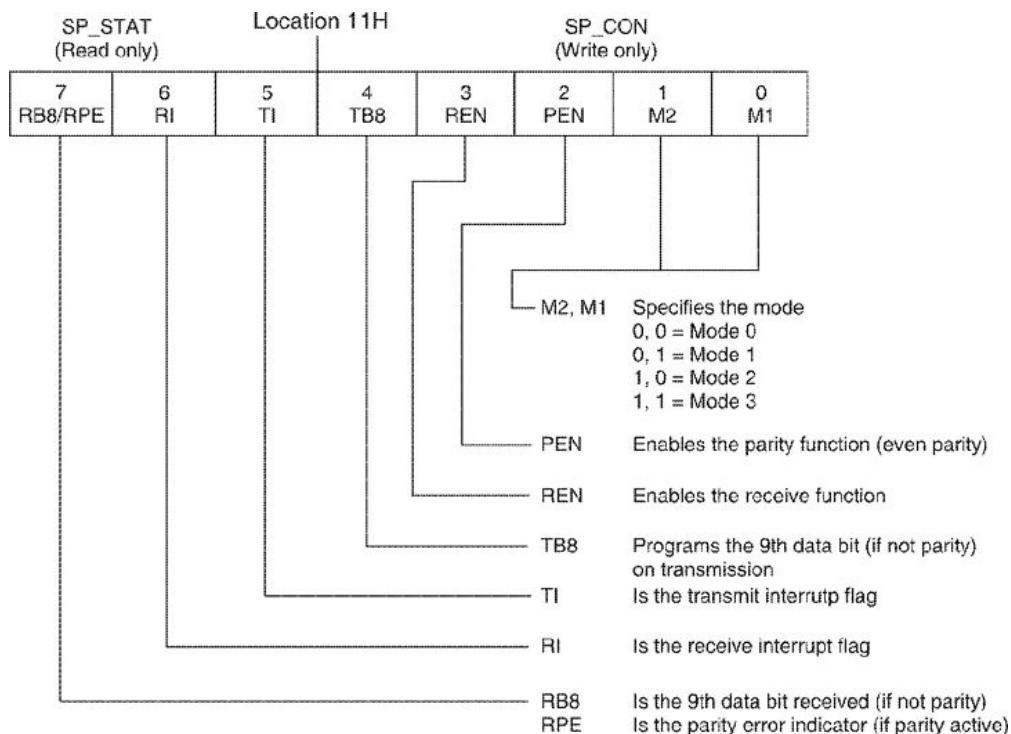


Figure 12.45 Serial port control/status register.

Baud rate

The BAUD_RATE register is an SFR at location 000EH. It is a 16-bit register. The MSB identifies the input frequency source. This register must be loaded sequentially with two bytes, the lower byte first.

If MSB = 1, Source = XTAL1 frequency

MSB = 0, Source = External frequency at T2CLK.

The maximum speed of XTAL1 frequency is 12 MHz and the maximum T2CLK speed is one transition every two state times, i.e. 2 MHz at 12 MHz XTAL1 frequency.

The 15-bit unsigned integer in BAUD_RATE register (after discarding the MSB) is used in the following way to calculate the baud rate.

$B = \text{BAUD_RATE register contents (15 bits)} + 1$ (for XTAL1 frequency)

$B = \text{BAUD_RATE register contents (15 bits)}$ (for T2CLK frequency)

Using XTAL1 frequency:

Mode 0: Baud rate = XTAL1 frequency/(4 \square B)

Other modes: Baud rate = XTAL1 frequency/(64 \square B)

Using T2CLK frequency:

Mode 0: Baud rate = T2CLK frequency/B

Other modes: Baud rate = T2CLK frequency/(16 \square B)

EXAMPLE 12.4

Figure 12.46 shows two 8096-based systems X and Y, interfaced through serial ports. Both systems are working on 12 MHz frequency which is also used for serial I/O. Let us assume that system X wishes to send 100 bytes to system Y in mode 1 at baud rate 9600. Explain the sequence of events in the two systems.

Solution:

Mode1 baud rate = XTAL1 frequency/(64 \square B)

i.e. $9600 = 12 \square 10^6 / (2^6 \square B)$

or $B = 12 \square 10^6 / (2^6 \square 9600)$

$\square 20$

15 bit contents in BAUD_RATE register = B - 1 =
0000000000010011B

MSB of BAUD_RATE register = 1 for XTAL1 frequency

Therefore, BAUD_RATE register = 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1B = 8013H

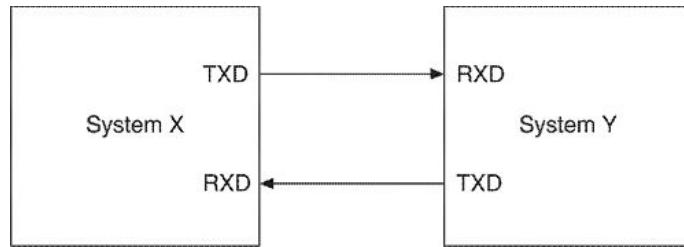


Figure 12.46 Serial input–output: an example.

Now the following will be the sequence of events in the two systems:

System X

Main Program

- Enable SIO interrupt by making Interrupt

Interrupt Mask Register bit 6 = 1

IMR = 01000000B = 40H

- Load BAUD_RATE register (000EH) register (000EH) with
with 13H

- Load BAUD_RATE register (000EH) register (000EH) with
with 80H

- Set IOC1.5 = 1 to enable P2.0 for TXD

SP_CON = 00000001 = 01H
01H

- Load SP_CON by 01H

- COUNT \square 100

- Load SBUF(TX) with the first byte

—Transmission starts

—

—

SIO Interrupt occurs

—ISR for SIO

—Read SP_STAT

Branch to ISR for TI

—ISR for TI

COUNT \square COUNT – 1

If COUNT > 0 Then

Load SBUF (TX)

System Y

Main Program

- Enable SIO interrupt by

Mask Register bit 6 = 1

IMR = 01000000B = 40H

- Load BAUD_RATE

13H

- Load the BAUD_RATE
80H

SP_CON = 00000001=

- Load SP_CON by 01H

—

—

SIO Interrupt occurs

—ISR for SIO

—Read SP_STAT

Branch to ISR for RI

—ISR for RI

Load SBUF(RX)
to memory

Enable Interrupt

Return

```
Enable Interrupt  
Return  
Else Return
```

12.14 MULTIPROCESSOR COMMUNICATIONS

Many complex applications may require more than one 8096 microcontroller working in interacting modes. The communication between these microcontrollers is performed using serial I/O mode 2 and mode 3 in the following way.

- When the master processor wants to send a block of data to one or more slave processors, it first sends the 11-bit frame which contains 8 bits for address and the 9th bit = 1 using mode 2.
- Each slave receives the frame (mode 2) and is interrupted since the 9th bit = 1. The ISR will examine the address information. The slave which is addressed, switches to mode 3 for receiving data frames, whereas the other slaves remain in mode 2 and continue what they were doing. The 9th bit in the data frame is 0. This bit differentiates between the address and the data frames.
- The master then sends the data bytes with the 9th bit = 0.

EXAMPLE 12.5

Figure 12.47 shows three 8096-based systems in the master-slave configuration. The master has been designated the address A whereas the slaves X, Y and Z have been designated the addresses B, C and D in hexadecimal. The master system wishes to send 50 bytes stored in memory to one of the slaves, say Z. Explain the sequence of operations.

Solution:

The following will be the sequence of operations.

- (a) The baud rate for communication needs to be decided in the beginning and the BAUD_RATE register will be loaded. For 9600 baud rate with XTAL1 frequency, the value 8013H will be loaded. IOC1.5 should also be set to 1 to enable P2.0 for TXD.
- (b) SP_CON in the master system will be
M2 M1 = 10, PEN = 0, REN = 1, TB8 = 1
i.e. SP_CON = 0 0 0 1 1 0 1 0 = 1AH

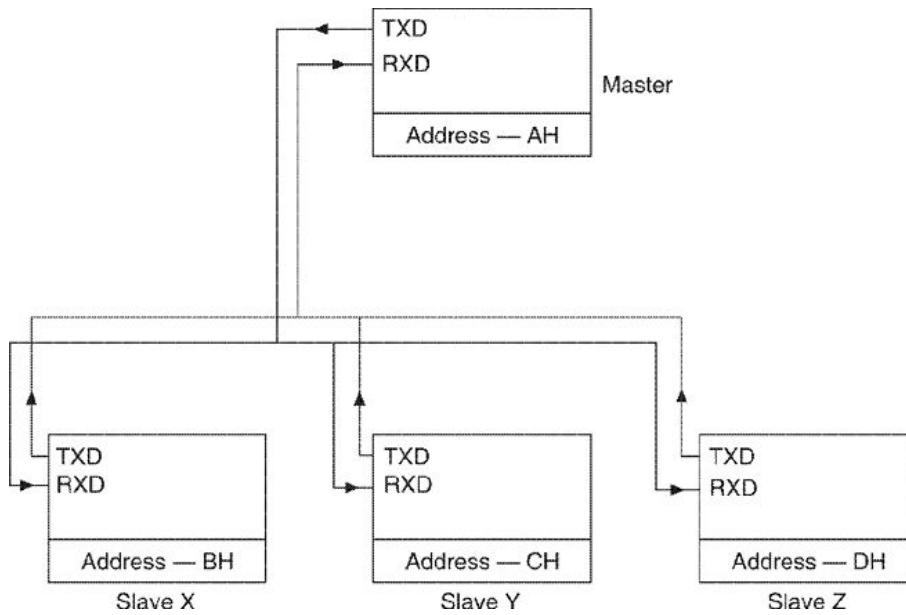


Figure 12.47 Multiprocessor communication: an example.

- (c) SP_CON in the slave register will be
 $M2\ M1 = 10, PEN = 0, REN = 1, TB8 = 0$
 i.e. $SP_CON = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0 = 0AH$
- (d) The SIO interrupt must be enabled in the master as well as the slaves by making the Interrupt Mask Register bit 6 = 1
 i.e. $IMR = 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0B = 40H$
- (e) The master sends the address frame. It is left to the designer to work out the handshaking protocol. Let us assume that the following address frame is being used.

1	Sender address(4 bits)	Receiver address(4 bits)	TB8	1
Start bit				Stop bit

In our example, the address frame will be

1 1010 1101 1 1

- (f) All the slaves receive the address frame. The SIO interrupt is caused in all the slaves since RB8 = 1.

SIO ISR in slave systems

Read SP_STAT

Branch to RI-ISR

Compare Receiver address with System address

If not equal then enable interrupt

Return

Else switch to mode 3 by modifying SP_CON.

(The modified SP_CON for slave Z will be – M2 M1 = 11, PEN = 0, REN = 1, TB8 = 0. Thus SP_CON will be 0 0 0 0 1 0 1 1 = 0BH)

Enable Interrupt

Return

- (g) The master system modifies SP_CON to make the 9th bit = 0.
Now

M2 M1 = 11, PEN = 0, REN = 0, TB8 = 0

i.e. SP_CON = 0 0 0 0 0 0 1 1 = 03H

Note: M2 M1 = 10 will also work. The master can send data in mode 2 or mode 3 as the data frame formats are the same.

- (h) The master sends the data byte by loading SBUF (TX) with the data byte. It will cause SIO interrupt at the end of the transmission.

SIO ISR in master system

Read SP_STAT

Branch to TI ISR

TI ISR in master

Load the next byte in SBUF(TX)

Enable Interrupt

Return

- (i) All the slaves will receive the data frame. Since the 9th bit = 0, the slaves in mode 2 will not be interrupted. The slave Z in mode 3 will be interrupted with SIO interrupt.

SIO ISR in slave Z

Read SP_STAT

Branch to RI ISR

RI ISR in slave Z

Read the data byte.

Check if it is the last byte.

If yes then enable interrupt

Return

Else store the data byte.

Enable interrupt

Return

Note: The slave system should get some indication regarding the end of transmission. It can be achieved in two ways.

1. The master can send as first byte the number of bytes to be transferred.
2. The master can send the end of message byte whose code is pre-decided.

12.15 ANALOG-TO-DIGITAL-CONVERTER

Some versions of the 8096 (the 8097, the 8397, the 8095, the 8395) contain 4/8 channel ADC which provides the 10-bit result. The ADC works on the successive approximation principle and takes 88 state times for conversion in the case of the 8096 BH. The reference voltage and the analog power supply voltage V_{REF} should be held at $(5.0 + 0.5)$ V. The analog input voltage must be in the range of 0 to V_{REF} . The 8096 contains the sample-and-hold circuit. The sampling window is open for four state times and this time is included in conversion time. At 12 MHz clock, the conversion time comes to 22 ms. Eight analog channels are represented on eight analog input pins (ACH0–ACH7) which are shared with Port 0. ACH7 can also be used as an external interrupt. This can be programmed by setting bit 1 of the I/O control register 1 to 1.

The conversion is initiated by the AD_COMMAND register (Figure 12.48) (an SFR at location 02H) which selects the channel to be converted and specifies when the conversion should start. It is possible to initiate the conversion either immediately after writing the command or after some time when the trigger from HSO is received. Locations 02H and 03H provide the result of the conversion. The format of the AD_RESULT SFR is shown in Figure 12.49.

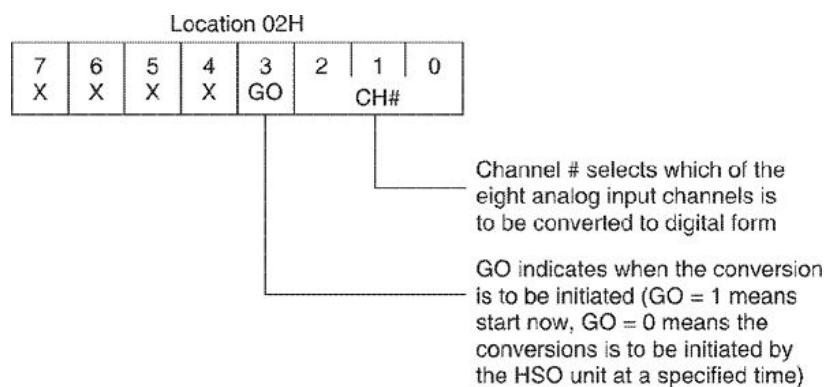


Figure 12.48 AD_COMMAND register.

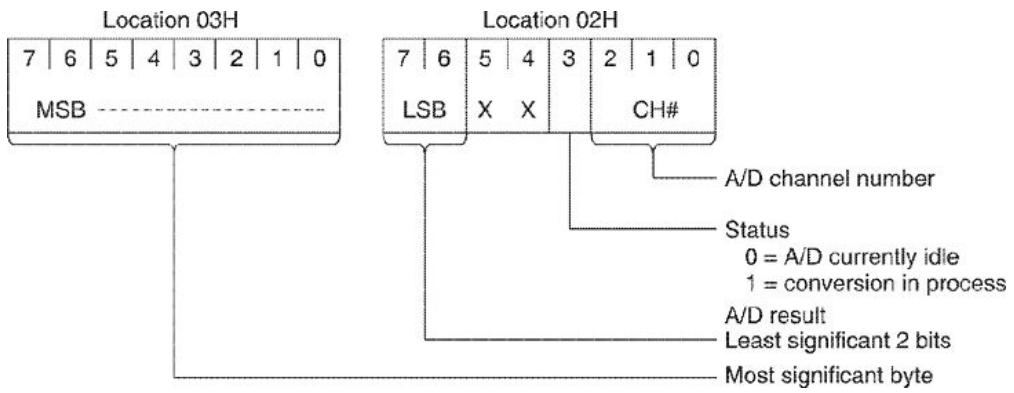


Figure 12.49 The AD_RESULT register.

12.16 ANALOG OUTPUT

There is a Pulse Width Modulation (PWM) facility which can be used to convert digital data to corresponding analog signals. The analog output is provided on pin 39/13 of the 8096. This is shared with P2.5. The PWM circuit (Figure 12.50) contains a register, a comparator and a counter. The register is loaded with a number and the counter is reset and initiated.

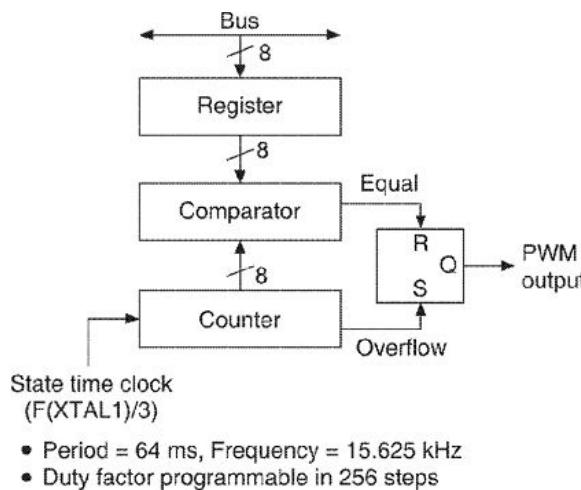


Figure 12.50 Pulse width modulated (D/A) output.

When Counter value = 0 ; PWM output = 1

When Counter value = Register value ; PWM output = 0

When Counter value overflows ; PWM output = 1

The output waveforms for different values stored in the register are shown in Figure 12.51. The pulse width can be changed by changing the number in the register. The output waveforms are variable duty cycle pulses which repeat themselves every 256 state times. The PWM_CONTROL register is an SFR at location 17H. There are several types of motors which require the PWM waveform for efficient

operations. The waveform can also be integrated using the buffer and the integrator to produce a dc voltage which will be proportional to the number in the PWM_CONTROL register. The bit 0 of the I/O Control Register 1 is used to select either the PWM output or P2.5. If bit = 1, select PWM otherwise select P2.5.

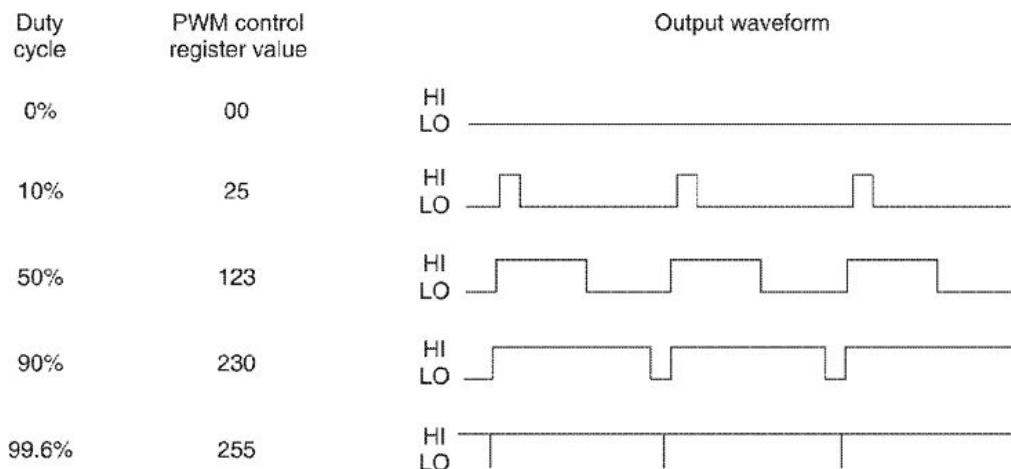


Figure 12.51 Typical PWM outputs.

EXAMPLE 12.6

Temperature is required to be monitored at eight locations in an air-conditioned railway coach. Based on the difference between the set point and the average of eight temperature values, the flow of air is regulated by 25%, 50%, 75% or 100% of the full-scale value. Explain the sequence of operations.

Solution:

The application requires eight analog input channels. Thus the 8097/8397 version of the 8096 will be used. The PWM output is connected to flow control valve through the buffer and the integrator. The complete schema is shown in Figure 12.52. The following will be the sequence of operations:

- Enable A/D conversion interrupt through Interrupt Mask Register bit 1 = 1.

Interrupt Mask Register = 0 0 0 0 0 0 1 0 = 02H

Load Interrupt Mask Register by 02H

- Set bit 0 of IOC1 to enable P2.5 as PWM output.

Load IOC1 by 01H

- Set a 16-bit register in Register File for SUM

SUM □ 0, LOOP □ 0

- For channel 0 to channel 7, do the following:

— Send A to D conversion command through AD_COMMAND register (location 02H)

AD_COMMAND register content

Channel 0 = x x x x 1 0 0 0 = 08H

Channel 1 = x x x x 1 0 0 1 = 09H

Channel 2 = x x x x 1 0 1 0 = 0AH

Channel 3 = x x x x 1 0 1 1 = 0BH

Channel 4 = x x x x 1 1 0 0 = 0CH

Channel 5 = x x x x 1 1 0 1 = 0DH

Channel 6 = x x x x 1 1 1 0 = 0EH

Channel 7 = x x x x 1 1 1 1 = 0FH

— ISR A/D Conversion

Read AD_RESULT LO (location 02H)

Read AD_RESULT HI (location 03H)

Put AD_RESULT in 16 bit TEMP register in the following way:

(TEMP)0 □ (AD_RESULT LO)6

(TEMP)1 □ (AD_RESULT LO)7

(TEMP)2-9 □ (AD_RESULT HI)0-7

Add TEMP to SUM

SUM □ SUM + TEMP

Determine if all channels have been scanned by checking the channel number in AD_RESULT (bits 0-2).

If Channel No. = 7, then

LOOP □ 1

Else LOOP □ 0

Enable Interrupt

Return

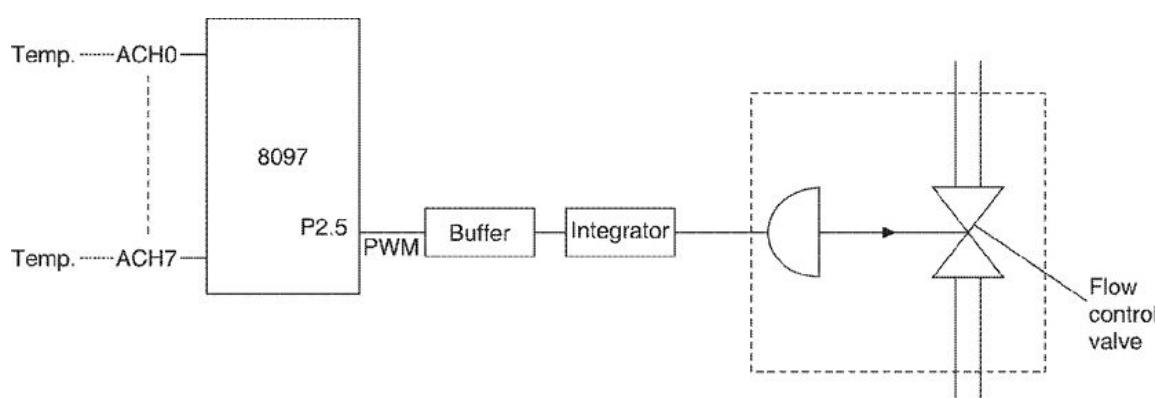


Figure 12.52 A to D and D to A: an example.

(e) Check if all the channels are scanned, if not then scan the remaining channels

If LOOP = 0 Then

Go to (d)

Else Go to (f)

(f) Calculate the average temperature

$$\text{AV_TEMP} = \text{SUM}/8$$

(g) Calculate the error and the error sign

$$\text{ERROR} = \text{SET_TEMP} - \text{AV_TEMP}$$

If AV_TEMP > SET_TEMP

Then E_SIGN = 0 (Error is negative)

Else E_SIGN = 1 (Error is positive)

(h) Calculate the % error and the % control voltage

$$\text{P_ERROR} = (\text{ERROR} \square 100)/\text{SET_TEMP}$$

If P_ERROR > 75 Then P_CONTROL = 100

Else If P_ERROR > 50 Then P_CONTROL = 75

Else If P_ERROR > 25 Then P_CONTROL = 50

Else If P_ERROR > 0 Then P_CONTROL = 25

Else P_CONTROL = 0

Following is assumed.

The setting of the current flow control valve is available in memory.

The PWM register output of 255 will get translated to 5 V through the buffer and the integrator which will fully open the valve.

Similarly 0 in the PWM register will result in 0 V output and fully close the valve.

(i) If E_SIGN = 0 (i.e. the average temperature is more than the set point)

Then

$$\text{New Flow Setting} = \text{Flow Setting} + (\text{P_CONTROL} \square 255)/100$$

If New Flow Setting > 255 Then New Flow Setting = 255

Else New Flow Setting = New Flow Setting

Else (i.e. the average temperature is less than the set point)

$$\text{New Flow Setting} = \text{Flow Setting} - (\text{P_CONTROL} \square 255)/100$$

If New Flow Setting < 0 Then New Flow Setting = 0
Else New Flow Setting = New Flow Setting
(j) Load the New Flow Setting in the PWM register.
(k) Replace the Flow Setting in memory by the New Flow Setting.
(l) Repeat from (c)

12.17 WATCHDOG TIMER

This feature is provided for the system to check itself and reset, if it is not functioning properly. The WATCHDOG register (an SFR at location 00AH) is a 16-bit counter which is incremented every state time. It pulls down the RESET line and resets the 8096 and other chips connected to the reset line, when the counter overflows. To prevent the timer overflow and consequently the system reset, the software should clear the counter periodically by writing 01EH followed by 0E1H, to WATCHDOG register location 00AH.

In many applications, the software may start behaving erroneously due to extraneous reasons like electrostatic discharge or any other hardware related problems. Since the microcontroller 8096 is used in real time control of the process plants, such errors may cause heavy damage. Thus the watch dog timer facility is very important and provides a means of graceful recovery if and when such problems occur.

During program development, the watchdog facility may have to be disabled otherwise the system will reset too many times. By holding the RESET pin at 2.0 to 2.5 V, the Watchdog Timer facility can be disabled.

12.18 CONCLUSION

The hardware details of the 8096 family of microcontrollers clearly indicate the trend to put all possible features in one chip. The software instruction set is equally powerful, as we shall see in the next chapter.

EXERCISES

1. Following memory chips need to be interfaced to the 8096:
6164 (Even)
6164 (Odd)
starting from the location 3000H.
Draw the schematic diagram of memory interface and work out the address decoding by 74LS138. How will the schematic diagram change in case of the write strobe mode?

2. The Centronics printer interface has been explained in Chapter 7. Develop the schema to interface the Centronics printer with the 8096.
3. Four thumbwheel switches are used in an application to input different values between 0 and 10. These switches basically output a dc voltage proportional to the value set. Interface the thumbwheel switches to the 8096 using the analog channels.
4. Extend the domain of Exercise No. 3 by interfacing four seven-segment LEDs to display the setting of thumbwheel switches.
5. Draw the schematic diagram and develop a flow chart to measure the pulse width, the frequency and the period of a signal using HSI, Timer and Interrupt system.
6. The conveyor belt system in an airport works in the following way:
 - (a) When switched on manually, a relay R1 is switched on after 5 seconds to initiate the circuit for internal checking.
 - (b) After 10 seconds, another relay R2 is switched on to initiate the first circuit block.
 - (c) The final circuit block is switched on after 20 seconds through relay R3. This starts the conveyor belt.
 - (d) To stop the conveyor belt, the above operations are performed in reverse order after the manual switch off command is given, i.e. relay R3, R2 and finally R1 are switched off.
 - (e) If there is any problem in the above operation sequence, then relay R4 is switched on to announce the problem and call the maintenance staff.Develop a schema using the 8096 to control the operation of the conveyor belt.

FURTHER READING

- 16 Bit Embedded Controller Handbook*, Intel Corporation, Santa Clara.
- Embedded Microcontrollers and Processors*, Intel Corporation, Santa Clara.
- Peripheral Design Handbook*, Intel Corporation, Santa Clara.

13

INTEL 8096 MICROCONTROLLER INSTRUCTION SET AND PROGRAMMING

13.1 INTRODUCTION

So far we have dealt with the hardware capabilities of the 8096 microcontroller. We have learnt that it has a unique architecture, quite different from the 8085 and also, to a large extent, different from the 8051. In the present chapter, we will deal with the programming of the 8096. We begin by describing the programmer's model, operand types and then the addressing modes. The instruction set in general would be covered, along with examples.

Having done the instruction sets and the programming models of the 8085, the 8086 and the 8051, we conclude that the types of instructions and their functions are more or less the same in different microprocessors.

13.2 PROGRAMMER'S MODEL OF THE 8096

A clear understanding of the hardware features of the 8096 is essential for being able to program it to interact with the external world. In Chapter 12 we dealt with the various hardware resources available in the 8096 in detail. However, the following resources would be used in the programming of the 8096.

- Memory
- Special Function Registers
- I/O Control and Status Registers

- Program Status Word

The users will have to refer to these resources quite often as they engage themselves in the exercise of writing software.

13.2.1 Memory

The 8096 contains 64 KB of addressable memory space. Figure 13.1 shows the memory map. The basic blocks are:

External memory or I/O	FFFFH
	4000H
Internal program storage ROM/EPROM or External memory	2080H
Reserved	2030H-207FH
Security key	2020H-202FH
Reserved	201CH-201FH
Self jump opcode (27H FEH)	201AH-201BH
Reserved	2019H
Chip configuration byte	2018H
Reserved	2012H-2017H
Interrupt vectors	
Port 4	2000H
Port 3	1FFFH
External memory or I/O	1FFEH
Internal RAM	0100H
Register file	00FFH
Stack pointer	
Special function registers	
Power down RAM (when accessed as data memory)	0000H

Figure 13.1 Memory map.

(a) On-chip RAM (Addresses 0000H to 00FFH)

- Special function registers (00H to 17H)
- Stack pointer (18H and 19H)
- Register file (1AH to EFH)
- Power down RAM (F0H to FFH)

This memory area is accessed as data memory and no code may be executed from this area. The program memory area of 00H to FFH is reserved for internal use of Intel development systems.

(b) Internal ROM in case the chip has on-chip ROM. If the chip does not have ROM, then the external memory is used (Addresses 2000H

to 3FFFH) containing:

- Interrupt vectors
- Factory test code
- Internal program storage

(c) Port 3 and Port 4 locations (Addresses 1FFEH to 1FFFH) used for Port 3 and Port 4 reconfigurations, if they are not used as address/data lines.

(d) External memory or I/O (Addresses 0100H to 1FFDH and 4000H to FFFFH).

When the 8096 is reset, the address 2080H is loaded to the program counter to give 8 KB of continuous memory.

13.2.2 Special Function Registers

The CPU communicates with other resources of the 8096 through Special Function Registers (SFRs) defined (Figure 13.2) in internal RAM space (00H to 19H). Through these SFRs, the CPU controls the various Timers, High Speed Input and Output Points, Interrupts, ADC, Stack and I/O ports.

19H	STACK POINTER	
18H		
17H		
16H	IOS1	
15H	IOS0	
14H		
13H	RESERVED	
12H		
11H	SP_STAT	
10H	IO PORT 2	
0FH	IO PORT 1	
0EH	IO PORT 0	
0DH	TIMER2 (HI)	
0CH	TIMER2 (LO)	
0BH	TIMER1 (HI)	
0AH	TIMER1 (LO)	
09H	INT_PENDING	
08H	INT_MASK	
07H	SBUF (RX)	
06H	HSI_STATUS	
05H	HSI_TIME (HI)	
04H	HSI_TIME (LO)	
03H	AD_RESULT (HI)	
02H	AD_RESULT (LO)	
01H	R0 (HI)	
00H	R0 (LO)	
	(When Read)	
		25
		24
		23
		22
		21
		20
		19
		18
		17
		16
		15
		14
		13
		12
		11
		10
		9
		8
		7
		6
		5
		4
		3
		2
		1
		0
	(When Written)	

Figure 13.2 Memory map of SFRs.

13.2.3 I/O Control and Status Registers

The four SFRs namely IOS0, IOS1, IOC0 and IOC1 are dedicated for I/O status and control. The SFRs IOS0 and IOC0 share the address 15H, whereas IOS1 and IOC1 share the address 16H. It is IOS0 and IOS1 when read and IOC0 and IOC1 when written (Figure 13.2).

- IO Status Register 0 (IOS0) contains information on the status of High Speed Output unit in terms of the current state of HSO points, Content Addressable Memory and Holding Register (Figure 13.3).

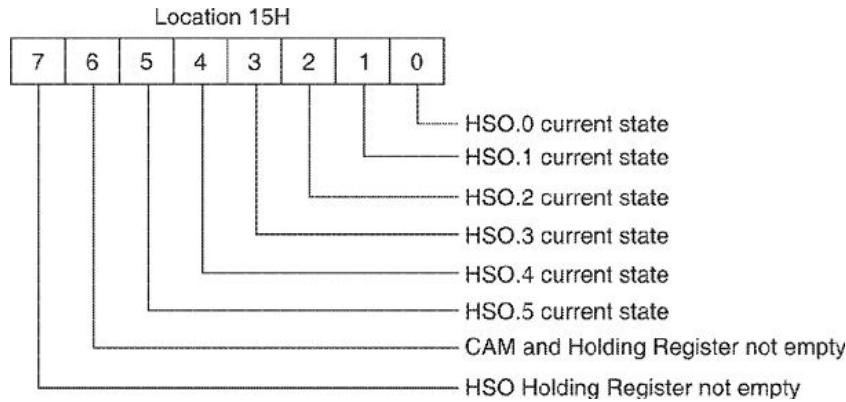


Figure 13.3 HSIO status register 0 (IOS0).

- IO Status Register 1 (IOS1) contains information on Timer 1 overflow, Timer 2 overflow, High Speed Input FIFO, High Speed Input Holding Register and Software Timers 0 to 3 (Figure 13.4).

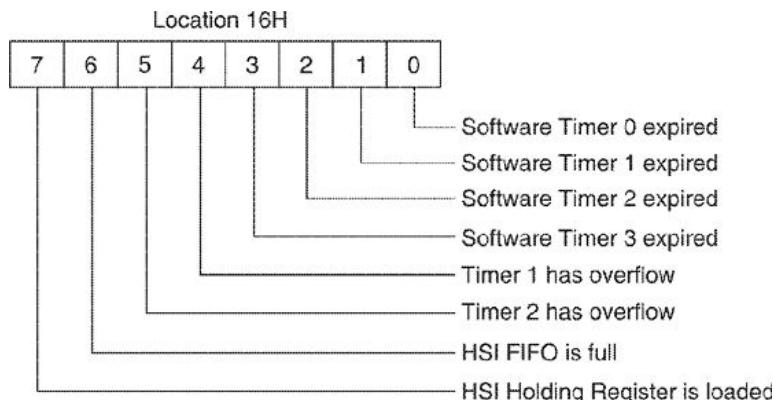


Figure 13.4 HSIO status register 1 (IOS1).

- IO Control Register 0 (IOC0) controls alternate functions of High Speed Input pins, and functions of Timer 2 like Clock Source, Reset Source, External Reset of Timer 2, and Timer 2 Reset Each Write (Figure 13.5).

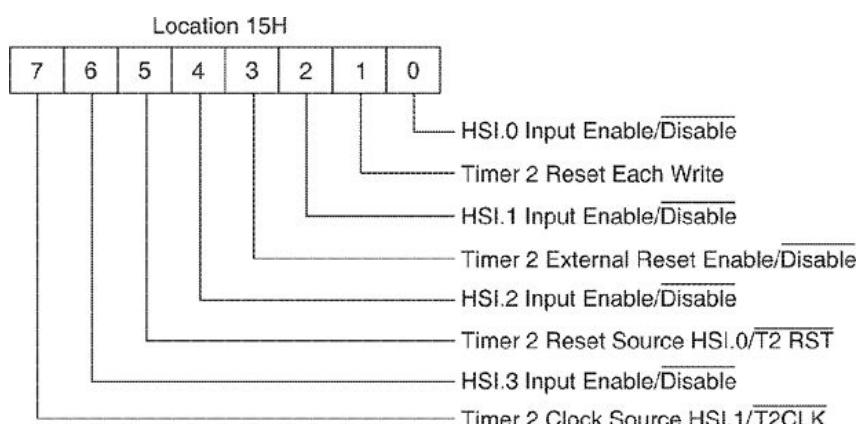


Figure 13.5 I/O control register 0 (IOC0).

- IO Control Register 1(IOC1) controls the alternate functions of Port 2 pins, Timer Interrupt and HSI interrupts (Figure 13.6).

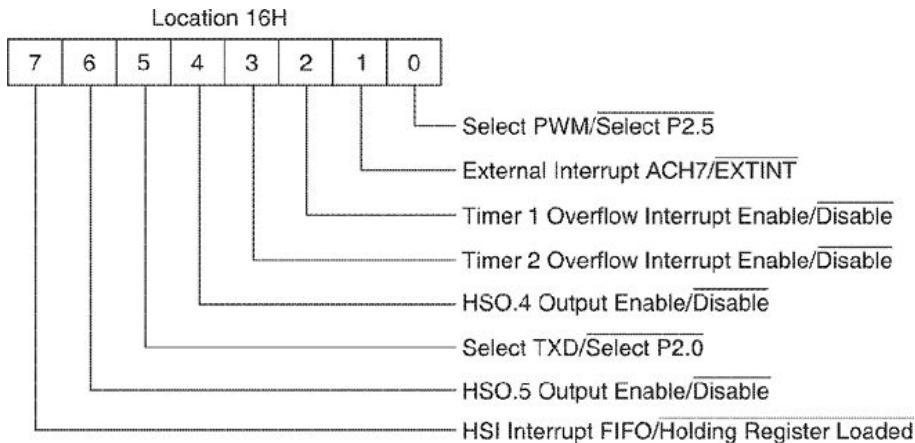


Figure 13.6 I/O control register 1 (IOC1).

These registers will be very frequently used while programming the 8096 in any real-time application environment.

13.2.4 Program Status Word

The Program Status Word (PSW) is the collection of boolean flags which retain information concerning the state of the user's program as well as the status of interrupt masking. The format of the PSW is shown below. The information in the PSW can be broken down into two basic categories, interrupt control and condition flag. The PSW can be saved in the system stack with a single operation (PUSHF) and restored in a like manner (POPF).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
Z	N	V	VT	C	—	I	ST							<Interrupt Mask Reg>

Interrupt mask register

The lower 8 bits of the PSW are used to individually enable/disable various sources of interrupts to the 8096. A logical 1 in these bit positions enables the servicing of the corresponding interrupt. These mask bits can be accessed as an 8-bit byte (INT_MASK register) at address 08H in the onboard register file. Note that the working of the Interrupt Mask Register in the 8096 is quite the opposite of that in the 8085 and that way it is not appropriate to call it Interrupt Mask Register.

It is basically an Interrupt Enable Register. The Interrupt Mask Register in the 8096 has been described under the Interrupts in Chapter 12.

Condition flags

The remaining bits in the PSW are set as side effects of instruction execution and can be tested by conditional jump instructions.

Z: The Z (Zero) flag is set to indicate that the operation generated a result equal to zero. For the add with carry (ADDC) and the subtract with borrow (SUBC) operations, the Z flag is cleared if the result is nonzero but is never set.

N: The N (Negative) flag is set to indicate that the operation generated a negative result. The N flag is set to the algebraically correct state even if the calculation overflows.

V: The V (Overflow) flag is set to indicate that the operation generated a result which is outside the range that can be expressed in the destination data type.

VT: The VT (Overflow Trap) flag is set whenever the V flag is set but can only be cleared by an instruction which explicitly operates on it, such as CLRVT or JVT instructions. The operation on the VT flag allows for the testing for a possible overflow condition at the end of a sequence of related arithmetic operations. This is normally more efficient than testing the V flag after each instruction.

C: The C (Carry) flag is set to indicate the state of the arithmetic carry from the most significant bit of the ALU for an arithmetic operation or state of the last bit shifted out of the operand for a shift. Arithmetic borrow after a subtract operation is the complement of the C flag (i.e. if the operation generated a borrow, then $C = 0$).

ST: The ST (Sticky bit) flag is set to indicate that, during a right shift, a 1 has been shifted into the C flag and then it has been shifted out. The ST flag may be used to control the rounding after a right shift.

I: The I (Global Interrupt Enable/Disable bit) is cleared by DI and set to 1 by EI instructions.

13.3 OPERAND TYPES

The 8096 architecture provides support for the following data types.

- Bytes
- Words
- Short Integers

- Integers
- Bits
- Double Words
- Long Integers

Bytes and short integers are 8-bit variables and can be placed anywhere in memory, without any alignment restrictions. Integer and Words are 16-bit variables. As a convention, 16-bit variables are aligned at even address boundaries in the address space, with lower byte at even address and higher byte being placed at the next higher address. The specified address is the address of the lower byte. The bits in the internal register file may be directly tested. However, the 8096 does not allow the direct addressing of bits unlike that in the 8051.

The double words and long integers are 32-bit variables used in shift or as dividend in 32 bit by 16 bit divide and product of 16 bit by 16 bit multiply. For these operations Double Word (Long Integer) must either reside in onboard register file of the 8096 or be aligned at an address which is divisible by four. The double word or long integer operand is addressed by the address of the least significant byte.

You must know that bytes, words and long words represent unsigned numbers, whereas short integers, integers and long integers represent signed numbers.

13.4 OPERAND ADDRESSING

Operands are accessed within the address space of the 8096 with one of the six basic addressing modes. The basic addressing modes are:

- Register direct
- Indirect
- Indirect with auto increment
- Immediate
- Short-indexed
- Long-indexed

Several other useful addressing operations can be achieved by combining these basic addressing modes with specific registers such as the ZERO register or the stack pointer.

Register direct references

The register direct mode is used to directly access a register from the 256 byte onboard register file. The register is selected by an 8-bit field within the instruction and the register address must conform to the alignment rules for the operand type. Depending on the instruction, up to three registers can take part in the calculation.

Instruction format

Mnemonic	Dest or Src1; Single operand register direct
Mnemonic	Dest, Src1; Two operand register direct
Mnemonic	Dest, Src1, Src2; Three operand register direct

where

Dest = Destination register

Src1, Src2 = Source register

Figure 13.7 shows the instruction execution in respect of the following register direct mode instructions.

1. LDB R2, R1; Load byte register R2 with the contents of the byte register R1.
 $(R2) \leftarrow (R1)$
2. ADD R6, R7, R8; Add word registers R7 and R8 and place the result in word register R6.
 $(R6) \leftarrow (R7) + (R8)$

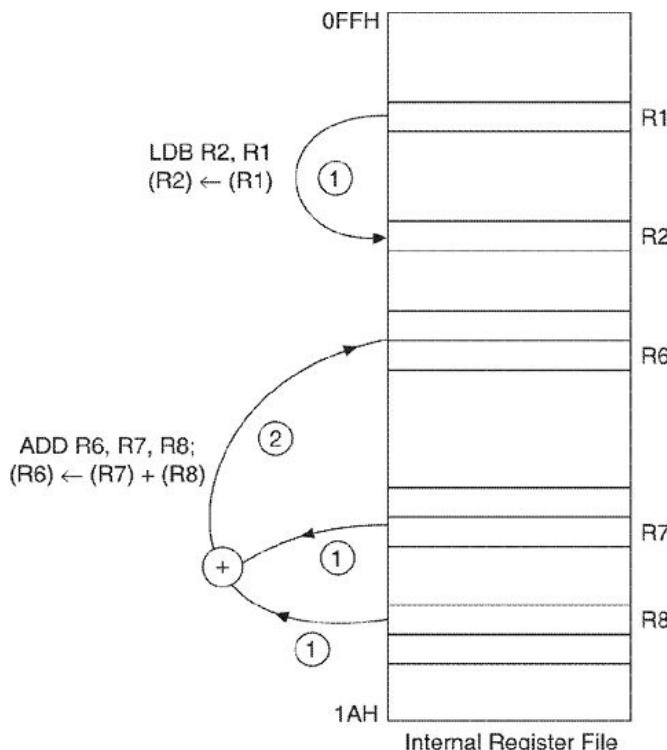


Figure 13.7 Register direct addressing: an example.

Indirect references

The indirect mode is used to access an operand by placing its address in a Word variable in the register file. The calculated address must conform to the alignment rules for the operand type. Note that the indirect address can refer to an operand anywhere within the address space of the 8096 including the register file. The register which contains the indirect address is selected by an 8-bit field within the instruction. An instruction can contain only one indirect reference, and the remaining operands of the instruction (if any) must be register direct references.

Instruction format

Mnemonic [addr]; Single operand indirect

Mnemonic Dest, [addr]; Two operand indirect

Mnemonic Dest, Src1, [addr]; Three operand indirect

where

Dest = Destination register

Src1 = Source register

addr = Word register used in computing the address of an operand

Figure 13.8 shows the execution of the following indirect mode instruction.

SUBB R1, (R5)
 $(R1) \square (R1) - ((R5))$

This instruction subtracts the contents of the memory location whose address is given in word register R5 from the contents of the byte register R1 and places the result in byte register R1.

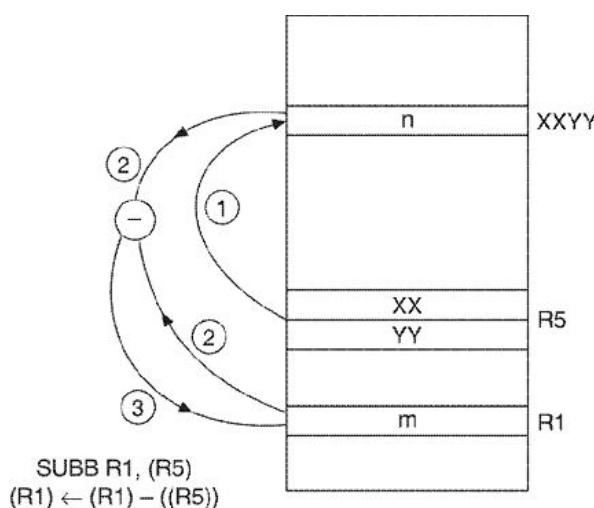


Figure 13.8 Indirect addressing: an example.

Indirect with auto-increment references

This addressing mode is the same as the indirect mode, except that the Word variable which contains the indirect address is incremented after it is used to address the operand. If the instruction operates on BYTES or SHORT INTEGERS, the indirect address variable will be incremented by 1; if the instruction operates on WORDS or INTEGERS, the indirect address will be incremented by 2.

Instruction format

Mnemonic [addr]+; Single operand indirect auto-increment

Mnemonic Dest, [addr]+; Two operand indirect auto-increment

Mnemonic Dest, Src1, [addr]+; Three operand indirect auto-increment

where

Dest = Destination register

Src1 = Source register

addr = Word register used in computing the address of an operand

Figure 13.9 shows the execution of the following indirect with auto-increment mode instruction.

SUBB R1, (R2) +
(R1) □ (R1) – ((R2)) ; (R2) □ (R2) + 1

This instruction subtracts the contents of memory location whose address is given in word register R2 from the contents of byte register R1 and places the result in byte register R1. It then increments the contents of the word register R2 by 1.

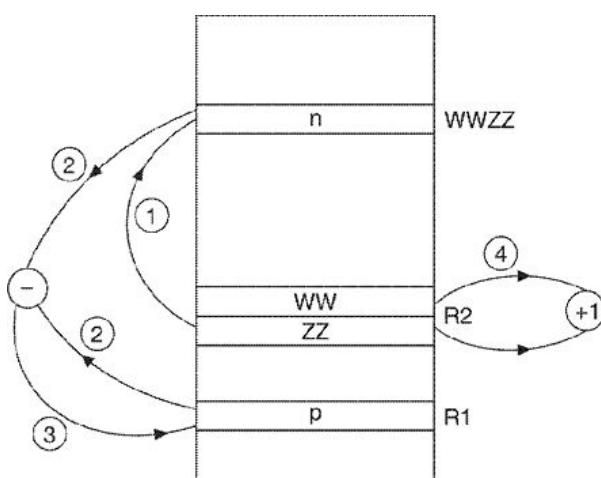


Figure 13.9 Indirect with auto-increment mode instruction: an example.

Immediate references

This addressing mode allows an operand to be taken directly from a field in the instruction. For operations on BYTE or SHORT INTEGER operands, this field is 8 bits wide, for operations on WORD or INTEGER operands, the field is 16 bits wide. An instruction can contain only one immediate reference and the remaining operand(s) must be register direct references.

Instruction format

Mnemonic	#value; Single operand immediate
Mnemonic	Dest, #value; Two operand immediate
Mnemonic	Dest, Src1, #value; Three operand immediate

where

value = immediate value

Dest = Destination register

Src1 = Source register

The execution of an instruction is illustrated below:

```
LD R1, #50  
(R1) □ 50  
MULB R5, #3  
(R5) □ (R5) □ 3
```

where R5 and R1 are word registers.

Indexed references

Two types of indexed addressing modes are possible—short indexed and long indexed. In short indexed addressing mode, an 8-bit field in the instruction selects a Word variable in the register file which is assumed to contain an address. A second 8-bit field in the instruction stream is sign-extended and summed with the Word variable to form the address of the operand which will take part in the calculation. Since the 8-bit field is sign-extended, the effective address can be up to 128 bytes before the address in the Word variable and up to 127 bytes after it. An instruction can contain only one short-indexed reference and the remaining operand(s) must be register direct references.

Instruction format

Mnemonic	Dest, offs[addr]; Two operand indexed (Long or Short)
Mnemonic	Dest, Src1, offs[addr]; Three operand indexed (Long or Short)

where

Dest = Destination register

Src1 = Source register

offs = Offset used in computing the address of an operand.

addr = Word register used in computing the address of an operand.

The long-indexed addressing mode is like the short-indexed mode except that a 16-bit field is taken from the instruction and added to the Word variable to form the address of the operand. No sign extension is necessary. An instruction can contain only one long-indexed reference and the remaining operand(s) must be register direct references.

Figure 13.10 shows the execution of the following long-indexed reference instruction.

DIVUB R3, 250[R7]

R3, R7—word register

250—offset

$(R3)(L) \square (R3)/M[(R7) + 250]$ —Quotient

$(R3)(H) \square (R3) \text{ MOD } M[(R7) + 250]$ —Remainder

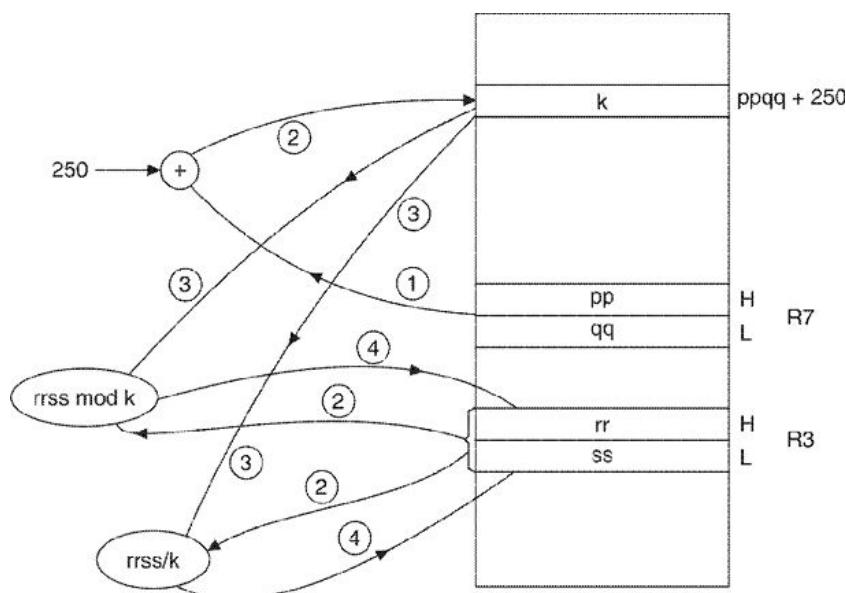


Figure 13.10 Long-indexed reference instruction: an example.

Figure 13.11 shows the execution of the following short-indexed reference instruction.

LDB R5, 20[R7]

R7—word register

R5—byte register

20—offset

$(R5) \square M[(R7) + 20]$

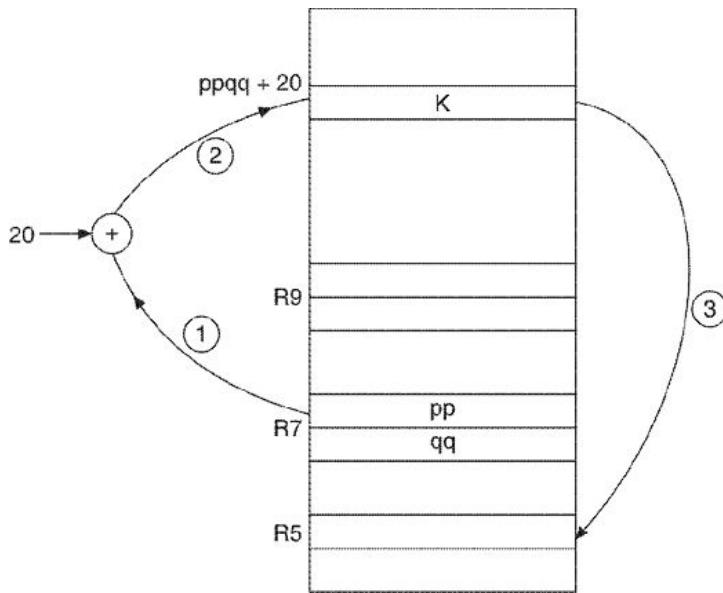


Figure 13.11 Short-indexed reference instruction: an example.

Zero register addressing

The first two bytes in the register file are fixed at ZERO by the 8096 hardware. In addition to providing a fixed source of the constant zero for calculations and comparisons, this register can be used as the Word variable in a long-indexed reference. This combination of the register selection and address mode allows any location in memory to be addressed directly.

Stack pointer register addressing

The stack pointer in the 8096 can be accessed as register 18H of the internal register file. In addition to providing for convenient manipulation of the stack pointer, this also facilitates the accessing of operands in the stack. The top of the stack, for example, can be accessed by using the stack pointer as the Word variable in an indirect reference. In a similar fashion, the stack pointer can be used in the short-indexed mode to access data within the stack.

13.5 MCS-96 ASSEMBLER DIRECTIVES

The utility of assembler directives has been already discussed in Chapter 2. Some directives like DB, DW, EQU, ORG, etc. which are common in different assemblers were also described. In Chapter 6, the assembler directives specific to the 8086 assembly language were covered. Here we shall cover the assembler directives with specific reference to MCS-96 macro assembler of Intel which runs on their Microcomputer Development Systems.

In MCS-96 assembler, the assembly program may be divided into a number of segments (similar to the 8086 assembler) namely Register Segment, Code Segment, Data Segment and Stack Segment. In Register Segment, a portion of memory is defined to function as the register. The Code Segment contains the executable instructions etc. The directives are:

RSEG	for	Register	Segment
CSEG	for	Code	Segment
DSEG	for	Data	Segment
SSEG	for	Stack	Segment

Following assembly program illustrates the RSEG and CSEG directives.

```

RSEG
AX:      DSW      1
AL:      EQU      AX:BYTE
AH:      EQU      (AX+1): BYTE
CSEG
LD       AX, 030H[0]
ADDB    AH, AL
STB     032H[0], AH
END

```

In the above program, the absolute location of the code and the registers is not specified. To specify the location of the data to the assembler, we have the directives REL, AT and ORG. (The default value of the locations from where RSEG and CSEG start are 01BH and 2080H respectively. This is a feature of MCS-96 Assembler and is based on the hardware feature of the 8096. REL and AT are the options available with CSEG, RSEG and DSEG directives. REL option specifies that the segment is relocatable. If a segment of the same type exists before, then the previous segment is continued as illustrated below.

```

A — DSEG REL
  VAR1:      DSB      1
  VAR2:      DSB      3      :
  :
B — CSEG REL      :
  :
C — DSEG REL

```

```

    VAR4:      DSB      1
    VAR5:      DCW      4080H
    VAR8:      DCB      55H
    :
D — CSEG REL      :
    :
    END

```

The assembler treats the data segments at A and C as lying in contiguous memory locations. Therefore, if VAR1 is stored at 0020H, then VAR4 will be stored at 0024H. Similarly, CSEG at B and D will be treated as present at contiguous memory locations. The REL option is the default and therefore, need not be specified if the segment is intended to be relocatable.

AT option specifies that the segment is absolute and begins at the address specified, as shown below.

```

RSEG AT 030H
BX:      DSW      1
CX:      DSW      1
CSEG AT 2100H
LD      BX, #0898
LD      CX, BX
END

```

In this case, BX is at 030H and CX is at 032H. Similarly, the machine codes are stored beginning from 2100H.

REL and AT are used along with the segment selection statements. On the other hand, ORG is used inside a segment. It is used to set the value of the location counter and can be used within an absolute segment or a relocatable segment.

```

CSEG
:
ORG 200H      :
:
END

```

The above code segment is a relocatable one. Therefore, the code of the first instruction following ORG 200H will be at 200H offset, from the base address of the CSEG segment. In case the code segment is an

absolute one, then the expression following ORG gives the absolute address of the following instruction.

```
CSEG AT 2100H      :  
      :  
      ORG      2300H  
      :  
      END
```

It is obvious that the address specified in ORG statement should not be less than at least the base address of the segment. Since the ORG directive does not create a new segment, it may only create gaps in the segment.

Falling in line with the Register Segment and the Code Segment are two more types of segments—the Data Segment and the Stack Segment. The Data Segment is a portion of the memory allocated in a RAM section. Variables belonging to this segment are used as memory referenced operands. Only the storage reservation can be specified in this segment. This segment is specified by the statement DSEG. Stack segment is also a portion of the memory allocated in the RAM section. However, neither code nor storage can be defined within this segment.

Inside the Register Segment and Data Segment, the user needs to reserve storage for program variables. The directives used for specifying the storage of program variables are classified as Storage Reservation Directives. These have been covered in Chapter 2.

DSB defines the storage for byte variables. DCB directive is used to define the byte storage and store an 8-bit value. No adjustment for boundaries is performed. DSW defines the storage for word variables. Using DCW, the word storage is defined and a 16-bit value is stored. Adjustment for word boundaries is performed. Similar is the case for DSL and DCL directives for long word (32 bit) with adjustments for long word boundaries. The storage reservation directives cannot appear in a code segment.

13.6 ACRONYMS USED

Several acronyms are used in the instructions and their descriptions, some of which are defined here.

breg: A byte register in the internal register file. Some confusion may occur as to whether this field refers to a source or a destination register; the register is therefore prefixed with an “S” or a “D”.

baop: A byte operand which is addressed by any of the addressing modes.

bit no.: A 3-bit field within an instruction op-code which selects one of the eight bits in a byte.

wreg.: A word register in the internal register file. If there is confusion as to whether this field refers to a source register or a destination register, it will be prefixed with an “S” or a “D”.

waop.: A word operand which is addressed by any of the addressing modes.

Ireg.: A 32-bit register in the internal register file.

cadd.: An address in the program code.

Flag settings

The modification to the flag setting is shown for each instruction. A check-mark () means that the flag is set or cleared as appropriate. A hyphen means that the flag is not modified. A one (1) or zero (0) indicates that the flag will be in that state after the instruction. An up arrow (↑) indicates that the instruction may set the flag if it is appropriate but will not clear the flag. A down arrow (↓) indicates that the flag may be cleared, but not set by the instruction.

Addressing modes

All the addressing modes, i.e. Direct, Immediate, Indirect (Normal and Auto increment), and Indexed (short and long) supported in the instruction set have been mentioned before.

13.7 DATA TRANSFER GROUP

This group contains the operations for the transfer of data from

- register to memory
- memory to register
- register to register
- immediate value to register

Both word as well as byte operations are supported.

Following types of instructions are possible in this group. *No flags are effected by these instructions.*

LD Load word: The value of the source (rightmost) WORD operand is stored into the destination (leftmost) operand.

DEST SRC
LD wreg, waop
(DEST) \square (SRC) Addressing modes—All

LDB Load byte: The value of the source (rightmost) BYTE operand is stored into the destination (leftmost) operand.

DEST SRC
LDB breg, baop
(DEST) \square (SRC) Addressing modes—All

LDBSE Load integer with short-integer: The value of the source (rightmost) BYTE operand is sign-extended and stored into the destination (leftmost) WORD operand.

DEST SRC
LDBSE wreg, baop
(low byte DEST) \square (SRC) Addressing modes—All
if (SRC) < 80H then
 (high byte DEST) \square 0
else
 (high byte DEST) \square FFH

LDBZE Load word with byte: The value of the source (rightmost) BYTE operand is zero-extended and stored into the destination (leftmost) WORD operand.

DEST SRC
LDBZE wreg, baop
(low byte DEST) \square (SRC) Addressing modes—All
(high byte DEST) \square 0

ST Store word: The value of the leftmost WORD operand is stored into the rightmost operand.

SRC DEST
ST wreg, waop
(DEST) \square (SRC) Addressing modes—Direct, Indirect, Indexed

STB Store byte: The value of the leftmost BYTE operand is stored into the rightmost operand.

SRC DEST
STB breg, baop
(DEST) \square (SRC) Addressing modes—Direct, Indirect, Indexed

Some examples are discussed below.

EXAMPLE 13.1

What will be the value in R2 after execution of the following instruction?

	LD	R1, #0500H
	LDB	R2, R1
R1:	DSW	1
R2:	DSB	1

Solution:

The first instruction will place 0500H in word register R1. At the end of the instruction,

$$\begin{aligned}(R1)(H) &= 05H \\ (R1)(L) &= 00H\end{aligned}$$

When the next instruction is executed, the lower byte of R1 is transferred to R2. Thus R2 will have 00H as value.

EXAMPLE 13.2

If the R2 word register contains 293FH and the memory location 293FH contains 87H, what will the value in word register R3 after the execution of instruction LDBSE R3, [R2]?

Solution:

$$\begin{aligned}\text{LDBSE R3, [R2]} \\ (R3) \square M [R2]\end{aligned}$$

The execution will bring the short integer in word register by preserving the sign of the value. Since 87H is a –ve number, the sign will be extended in higher byte. Thus,

$$\begin{aligned}(R3)(L) \text{ will have } 87H \\ (R3)(H) = FFH\end{aligned}$$

EXERCISES

1. Why is immediate addressing not possible in store operations?
2. The memory locations in the 8096 are shown below. Show how the data in these locations can be transferred to different registers using indexed addressing and indirect addressing.

Location address	Content
4004H	–
4003H	–
4002H	–
4001H	–
4000H	–

3004H	30H
3003H	25H
3002H	20H
3001H	70H
3000H	60H
Word register R4	—
Word register R3	—
Byte register R2	—
Word register R1	—

3. The contents of the memory locations 3000H, 3001H, 3002H, and 3003H are to be transferred to memory locations 4000H, 4001H, 4002H and 4003H respectively. Write the instructions, first using the indirect and then using the indexed addressing.

13.8 ARITHMETIC GROUP

In arithmetic group, the 8096 offers a variety of instructions to perform the operations of addition, subtraction, multiplication, and division.

Addition

In this group of instructions, we have the facility to add words, add bytes, add words with carry and add bytes with carry. It is possible to have two-operand instructions as well as three-operand instructions in add bytes and add words. Thus, the operations containing three different operands (the two operand sources and the destination) may also be performed.

It is also possible to increment a word or a byte operand.

In the instructions for addition with carry, i.e. ADDC and ADDCB, flags are affected in the following way.

Z	N	C	V	VT	ST
↓	√	√	√	↑	—

Following are the flags affected in all other addition instructions.

Z	N	C	V	VT	ST
√	√	√	√	↑	—

ADD (Two operands)—Add words: The sum of the two WORD operands is stored into the destination (leftmost) operand.

DEST SRC
ADD wreg, waop

$(DEST) \square (DEST) + (SRC)$ Addressing modes—All

ADD (Three operands)—Add words: The sum of the second and third WORD operands is stored into the destination (leftmost) operand.

DEST SRC1 SRC2

ADD Dwreg, Swreg, waop

$(DEST) \square (SRC1) + (SRC2)$ Addressing modes—All

ADDB (Two operands)—Add bytes: The sum of the two BYTE operands is stored into the destination (leftmost) operand.

DEST SRC

ADDB breg, baop

$(DEST) \square (DEST) + (SRC)$ Addressing modes—All

ADDB (Three operands)—Add bytes: The sum of the second and third BYTE operands is stored into the destination (leftmost) operand.

DEST SRC1 SRC2

ADDB Dbreg, Sbreg, baop

$(DEST) \square (SRC1) + (SRC2)$ Addressing modes—All

ADDC—Add words with carry: The sum of the two WORD operands and the carry flag (0 or 1) is stored into the destination (leftmost) operand.

DEST SRC

ADDC wreg, waop

$(DEST) \square (DEST) + (SRC) + (C)$ Addressing modes—All

ADDCB—Add bytes with carry: The sum of the two BYTE operands and the carry flag (0 or 1) is stored into the destination (leftmost) operand.

DEST SRC

ADDCB breg, baop

$(DEST) \square (DEST) + (SRC) + (C)$ Addressing modes—All

INC—Increment word: The value of the WORD operand is incremented by 1.

INC wreg

$(DEST) \square (DEST) + 1$ Addressing modes—Register

INCB—Increment byte: The value of the BYTE operand is incremented by 1.

INCB breg
 $(DEST) \square (DEST) + 1$ Addressing modes—Register

Subtraction

The subtraction operations can be performed with two-operand addresses as well as three-operand addresses on both words and bytes. There are also operations to subtract words as well as bytes with borrow.

Instructions for decrementing a word or byte operand are also provided.

In instructions for subtraction with borrow, i.e. SUBC and SUBCB, the flags are affected in the following ways

Z	N	C	V	VT	ST
↓	✓	✓	✓	↑	—

Following are the flags affected in all the other subtraction instructions.

Z	N	C	V	VT	ST
✓	✓	✓	✓	↑	—

SUB (Two operands)—Subtract words: The source (rightmost) WORD operand is subtracted from the destination (leftmost) WORD operand, and the result is stored in the destination. The carry flag is set as the complement of borrow.

DEST SRC
SUB wreg, waop
 $(DEST) \square (DEST) - (SRC)$ Addressing modes—All

SUB (Three operands)—Subtract words: The source (rightmost) WORD operand is subtracted from the second WORD operand, and the result is stored in the destination (the leftmost operand). The carry flag is set as the complement of borrow.

DEST SRC1 SRC2
SUB wreg, wreg, waop
 $(DEST) \square (SRC1) - (SRC2)$ Addressing modes—All

SUBB (Two operands)—Subtract bytes: The source (rightmost) BYTE is subtracted from the destination (leftmost) BYTE operand, and the result is stored in the destination. The carry flag is set as the complement of borrow.

DEST SRC
 SUBB breg, baop
 $(DEST) \square (DEST) - (SRC)$ Addressing modes—All

SUBB (Three operands)—Subtract bytes: The source (rightmost) BYTE operand is subtracted from the second BYTE operand, and the result is stored in the destination (the leftmost operand). The carry flag is set as the complement of borrow.

DEST SRC1 SRC2
 SUBB Dbreg, sbreg, baop
 $(DEST) \square (SRC1) - (SRC2)$ Addressing modes—All

SUBC—Subtract words with borrow: The source (rightmost) WORD operand is subtracted from the destination (leftmost) WORD operand. If the carry flag is clear, 1 is subtracted from the above result. The result replaces the original destination operand. The carry flag is set as the complement of borrow.

DEST SRC
 SUBC wreg, waop
 $(DEST) \square (DEST) - (SRC) - (1 - (C))$ Addressing modes—All

SUBCB—Subtract bytes with borrow: The source (rightmost) BYTE operand is subtracted from the destination (leftmost) BYTE operand. If the carry flag is clear, 1 is subtracted from the above result. The result replaces the original destination operand. The carry flag is set as the complement of borrow.

DEST SRC
 SUBCB breg, baop
 $(DEST) \square (DEST) - (SRC) - (1 - (C))$ Addressing modes—All

DEC—Decrement word: The value of the WORD operand is decremented by 1.

DEC wreg
 $(DEST) \square (DEST) - 1$ Addressing modes—Register

DECB—Decrement byte: The value of the BYTE operand is decremented by 1.

DECB breg
 $(DEST) \square (DEST) - 1$ Addressing modes—Register

Multiplication

The 8096 instructions support two- as well as three-operand multiplications. The operands may be integers (signed 16 bit numbers), short integers (signed 8 bit numbers), bytes, and words.

The flags affected in all multiplication instructions are

Z	N	C	V	VT	ST
—	—	—	—	—	?

In a multiplication operation, the sticky bit is undefined at the end of the execution of the instruction.

MUL (Two operands)—Multiply integers: The two INTEGER operands are multiplied using signed arithmetic and the 32-bit result is stored into the destination (leftmost) LONG-INTEGER operand.

DEST SRC
MUL lreg, waop
(DEST) □ (DEST) □ (SRC) Addressing modes—All

MUL (Three operands)—Multiply integers: The second and third INTEGER operands are multiplied using signed arithmetic and the 32-bit result is stored into the destination (leftmost) LONG INTEGER operand.

DEST SRC1 SRC2
MUL lreg, wreg, waop
(DEST) □ (SRC1) □ (SRC2) Addressing modes—All

MULB (Two operands)—Multiply short-integers: The two SHORT-INTEGER operands are multiplied using signed arithmetic and the 16-bit result is stored into the destination (leftmost) INTEGER operand.

DEST SRC
MULB wreg, baop
(DEST) □ (DEST) □ (SRC) Addressing modes—All

MULB (Three operands)—Multiply short-integers: The second and third SHORT-INTEGER operands are multiplied using signed arithmetic and the 16-bit result is stored into the destination (leftmost) INTEGER operand.

DEST SRC1 SRC2
MULB wreg, breg, baop
(DEST) □ (SRC1) □ (SRC2) Addressing modes—All

MULU (Two operands)—Multiply words: The two WORD operands are multiplied using unsigned arithmetic and the 32-bit result is stored into the destination (leftmost) DOUBLE-WORD operand.

DEST SRC	
MULU lreg, waop	
(DEST) □ (DEST) □ (SRC)	Addressing modes—All

MULU (Three operands)—Multiply words: The second and third WORD operands are multiplied using unsigned arithmetic and the 32-bit result is stored into the destination (leftmost) DOUBLE WORD operand.

DEST SRC1 SRC2	
MULU lreg, wreg, waop	
(DEST) □ (SRC1) □ (SRC2)	Addressing modes—All

MULUB (Two operands)—Multiply bytes: The two BYTE operands are multiplied using unsigned arithmetic and the WORD result is stored into the destination (leftmost) operand.

DEST SRC	
MULUB wreg, baop	
(DEST) □ (DEST) □ (SRC)	Addressing modes—All

MULUB (Three operands)—Multiply bytes: The second and third BYTE operands are multiplied using unsigned arithmetic and the WORD result is stored into the destination (leftmost) operand.

DEST SRC1 SRC2	
MULUB wreg, breg, baop	
(DEST) □ (SRC1) □ (SRC2)	Addressing modes—All

Division

The division operation can be performed using long integers, integers, short integers, bytes, double words and words. The dividend can be a 32-bit long integer or double word in case of divide integers and divide word instructions.

In case of instructions for divide integers (i.e. DIV) and divide short integers (i.e. DIVB), following are the flags affected.

Z	N	C	V	VT	ST
—	—	—	?	↑	—

In case of instructions for divide words (i.e. DIVU) and divide bytes (i.e. DIVUB) the flags are affected in the following way.

Z	N	C	V	VT	ST
—	—	—	✓	↑	—

Following are the instructions:

DIV—Divide integers: This instruction divides the contents of the destination LONG-INTEGER operand by the contents of the INTEGER operand, using signed arithmetic. The low-order word of the destination (i.e. the word with the lower address) will contain the quotient; the high-order word will contain the remainder.

DEST SRC	
DIV lreg, waop	
(low word DEST) \square (DEST)/(SRC)	Addressing modes
—All	
(high word DEST) \square (DEST) MOD (SRC)	

The above two operations are performed concurrently.

DIVB—Divide short-integers: This instruction divides the contents of the destination INTEGER operand by the contents of the source SHORT-INTEGER operand, using signed arithmetic. The low order byte of the destination (i.e. the byte with the lower address) will contain the quotient; the high-order byte will contain the remainder.

DEST SRC	
DIVB wreg, baop	
(low byte DEST) \square (DEST)/(SRC)	Addressing modes—
All	
(high byte DEST) \square (DEST) MOD (SRC)	

The above two operations are performed concurrently.

DIVU—Divide words: This instruction divides the contents of the destination DOUBLE-WORD operand by the contents of the source WORD operand, using unsigned arithmetic. The low-order word will contain the quotient; the high-order word will contain the remainder.

DEST SRC	
DIVU lreg, waop	
(low word DEST) \square (DEST)/(SRC)	Addressing modes—
All	
(high word DEST) \square (DEST) MOD (SRC)	

The above two operations are performed concurrently.

DIVUB—Divide bytes: This instruction divides the contents of the destination WORD operand by the contents of the source BYTE operand, using unsigned arithmetic. The low-order byte of the destination (i.e. the byte with the lower address) will contain the quotient; the high-order byte will contain the remainder.

DEST SRC	
DIVUB wreg, baop	
(low byte DEST) \square (DEST)/(SRC)	Addressing modes
—All	
(high byte DEST) \square (DEST) MOD (SRC)	

The above two operations are performed concurrently.

Comparison

It is possible to compare two bytes or two words. The operation performed is subtraction and the flags are set as a result of this. The flags affected are

Z	N	C	V	VT	ST
✓	✓	✓	✓	↑	—

CMPB—Compare bytes: The source (rightmost) BYTE operand is subtracted from the destination (leftmost) BYTE operand. The flags are altered but the operands remain unaffected. The carry flag is set as the complement of borrow.

DEST SRC	
CMPB breg, baop	
(DEST) – (SRC)	Addressing modes—All

CMP—Compare words: The source (rightmost) WORD operand is subtracted from the destination (leftmost) WORD operand. The flags are altered but the operands remain unaffected. The carry flag is set as the complement of borrow.

DEST SRC	
CMP wreg, waop	
(DEST) – (SRC)	Addressing modes—All

Negation

It is possible to negate or change the sign of a WORD operand or a BYTE operand. Following are the flags affected.

Z	N	C	V	VT	ST
✓	✓	✓	✓	↑	—

The operations are described below.

NEG—Negate integer: The value of the INTEGER operand is negated.

NEG wreg

(DEST) $\square -$ (DEST) Addressing modes—Register

NEGB—Negate short-integer: The value of the SHORT-INTEGER operand is negated.

NEGB breg

(DEST) $\square -$ (DEST) Addressing modes—Register

Some examples are discussed below.

EXAMPLE 13.3

The memory locations and contents are given below. Write instructions to add the contents of locations 3005H to 3008H to R1 and place the result in register R4.

<i>Memory location</i>	<i>Contents</i>
3009H	01H
3008H	15H
3007H	19H
3006H	09H
3005H	02H
Byte register R1	05H
Word register R2	—
Byte register R3	06H
Word register R4	—

Solution:

LD	R2, #3005H	; R2 is word register
ADDB	R4, R1, [R2]	; (R4) \square (R1) + M [3005]
ADDCB	R4, 1[R2]	; (R4) \square (R4) + (C) + M [3005+1]
ADDCB	R4, 2[R2]	; (R4) \square (R4) + (C) + M [3005+2]
ADDCB	R4, 3[R2]	; (R4) \square (R4) + (C) + M [3005+3]

EXAMPLE 13.4

As in Example 13.3, if the following instructions are executed, what will be the contents of register R3 at the end?

LD	R2, #3005H
SUBB	R4, R1, [R2]
SUBCB	R4, 1[R2]
ADDB	R4, 2[R2]
SUBB	R4, 3[R2]
ADDB	R3, R4

Solution:

- Instruction
1. $(R2) = 3005H$
 2. $(R4) = (R1) - M[3005]$
 $= 05H - 02H = 03H, (C) = 1$
 3. $(R4) = (R4) - M[3005 + 1] - (1 - (C))$
 $= 03H - 09H - 0 = FAH, (C) = 0$
 4. $(R4) = (R4) + M[3005 + 2]$
 $= FAH + 19H = 13H$
 5. $(R4) = 13H - M[3008]$
 $= 13H - 15H = FEH, (C) = 0$
 6. $(R3) = 06H + FEH = 04H$

EXAMPLE 13.5

The memory locations and contents of registers are given below. Write a program to multiply the contents of locations 4120H to 4123H by 2 and add to the contents of the word register R3.

<i>Memory location</i>	<i>Contents</i>
4123H	15H
4122H	03H
4121H	02H
4120H	10H
Word register R1	—
Byte register R2	—
Word register R3	0005H
Word register R4	—

Solution:

LDB	R2, #02H
LD	R1, #4120H
LD	R4, #0000H
MULB	R4, R2, [R1] ; (R4) \square (R2) \square M[4120]

ADD	R3, R4	; (R3) \square (R3) + (R4)
MULB	R4, R2, 1[R1]	; (R4) \square (R2) \square M[4121]
ADDC	R3, R4	; (R3) \square (R3) + (R4) + (C)
MULB	R4, R2, 2[R1]	; (R4) \square (R2) \square M [4122]
ADDC	R3, R4	; (R3) \square (R3) + (R4) + (C)
MULB	R4, R2, 3[R1]	; (R4) \square (R2) \square M [4123]
ADDC	R3, R4	; (R3) \square (R3) + (R4) + (C)

EXAMPLE 13.6

Four integer numbers X, Y, Z and W are stored in consecutive memory locations 2301H, 2302H, 2303H and 2304H. Calculate

$$P = \frac{(X \times Y) + (Z \times W)}{(X - Y)}$$

Store the result in word register R4 [Quotient in R4(L) and Remainder in R4(H)].

Solution:

LD	R2, #2301H	
LDB	R3, [R2]	; (R3) \square X
MULB	R4, R3, 1[R2]	; (R4) \square X \square Y (R4 is word reg)
LDB	R5, 2[R2]	; (R5) \square Z
MULB	R6, R5, 3[R2]	; (R6) \square Z \square W (R6 = word register)
ADD	R4, R6	; (R4) \square (X \square Y) + (Z \square W)
SUBB	R3, 1[R2]	; (R3) \square X - Y
DIVB	R4, R3	; (R4)(L) \square (R4)/(R3)
		; (R4)(H) \square (R4) MOD (R3)

EXERCISES

1. In Example 13.3, what will be the contents of R3 after the execution of the following instructions?

LD	R2, #3005H
SUBB	R3, R1, [R2]
ADDB	R3, 3[R2]

2. Five numbers are stored in consecutive locations starting from 3100H. Write a program to square each number and then divide by 5. Store the resultant numbers in memory locations starting from 3200H.

3. Write a program to calculate $\exp(x)$ where x is any integer number equal to or more than zero. The memory location 3500H contains the value of x . Calculate only up to 4 terms.

13.9 LOGICAL GROUP

In this group, three logical operations, i.e. AND, OR and XOR are supported. All the operations can be performed either on word or on byte operands. The AND (word operand and byte operand) instructions have three-operand versions of instructions, whereas others are basically two-operand instructions.

In addition, there are instructions like NOT and NOTB (for complement) which are based on single-operand register.

In all the logical instructions the following are the flags affected.

Z	N	C	V	VT	ST
✓	✓	0	0	—	—

AND (Two operands)—AND words: The two-WORD operands are ANDed, the result has a 1 only in those bit positions where both the operands have a 1, with 0s in all other bit positions. The result is stored into the destination (leftmost) operand.

DEST SRC	
AND wreg, waop	
(DEST) □ (DEST) AND (SRC)	Addressing modes—All

AND (Three operands)—AND words: The second and third WORD operands are ANDed, the result having a 1 only in those bit positions where both operands have a 1, with 0s in all other bit positions. The result is stored into the destination (leftmost) operand.

DEST SRC1 SRC2	
AND Dwreg, Swreg, waop	
(DEST) □ (SRC1) AND (SRC2)	Addressing modes—All

ANDB (Two operands)—AND bytes: The two BYTE operands are ANDed, the result has a 1 only in those bit positions where both operands have a 1, with 0s in all other bit positions. The result is stored into the destination (leftmost) operand.

DEST SRC	
ANDB breg, baop	
(DEST) □ (DEST) AND (SRC)	Addressing modes—All

ANDB (Three operands)—AND bytes: The second and third BYTE operands are ANDed, the result has a 1 only in those bit positions where both operands have a 1, with 0s in all other bit positions. The result is stored into the destination (leftmost) operand.

DEST	SRC1	SRC2	
ANDB	Dbreg,	Sbreg,	baop
(DEST)	□	(SRC1)	AND (SRC2)
			Addressing modes—All

OR—OR words: The source (rightmost) WORD operand is ORed with the destination (leftmost) WORD operand. Each bit is set to a 1, if the corresponding bit in either the source operand or the destination operand is a 1. The result replaces the original destination operand.

DEST	SRC	
OR	wreg,	waop
(DEST)	□	(DEST) OR (SRC)
		Addressing modes—All

ORB—OR bytes: The source (rightmost) BYTE operand is ORed with the destination (leftmost) BYTE operand. Each bit is set to a 1, if the corresponding bit in either the source operand or the destination operand is a 1. The result replaces the original destination operand.

DEST	SRC	
ORB	breg,	baop
(DEST)	□	(DEST) OR (SRC)
		Addressing modes—All

XOR—Exclusive-OR words: The source (rightmost) WORD operand is XORed with the destination (leftmost) WORD operand. Each bit is set to a 1, if the corresponding bit in either the source operand or the destination operand is a 1, but not both. The result replaces the original destination operand.

DEST	SRC	
XOR	wreg,	waop
(DEST)	□	(DEST) XOR (SRC)
		Addressing modes—All

XORB—Exclusive-OR bytes: The source (rightmost) BYTE operand is XORed with the destination (leftmost) BYTE operand. Each bit is set to a 1, if the corresponding bit in either the source operand or the destination operand is a 1, but not both. The result replaces the original destination operand.

DEST	SRC	
XORB	breg,	baop
(DEST)	□	(DEST) XOR (SRC)
		Addressing modes—All

NOT—Complement word: The value of the WORD operand is complemented: each 1 is replaced with a 0, and each 0 with a 1.

NOT wreg
(DEST) □ NOT (DEST) Addressing modes—Register

NOTB—Complement byte: The value of the BYTE operand is complemented: each 1 is replaced with a 0, and each 0 with a 1.

NOTB breg
(DEST) □ NOT (DEST) Addressing modes—Register

EXAMPLE 13.7

Originally the contents of registers R1, R2, R3 and R4 were as follows:

$$(R1) = FFH, (R2) = 19H, (R3) = 00H, (R4) = 0FH$$

Find the resultant byte in R4 after the execution of the following instructions.

1.	ANDB	R1, R2
2.	ORB	R1, R4
3.	XORB	R1, R3
4.	ANDB	R4, R1

Solution:

Originally,

$$\begin{aligned}(R1) &= 11111111 \\(R2) &= 00011001 \\(R3) &= 00000000 \\(R4) &= 00001111\end{aligned}$$

After the execution of

$$\begin{array}{ll} \text{First instruction: } & (R1) = 00011001 \\ \text{Second instruction: } & (R1) = 00011111 \\ \text{Third instruction: } & (R1) = 00011111 \\ \text{Fourth instruction: } & (R4) = 00001111 \end{array}$$

Thus R4 will contain 0FH at the end of the fourth instruction.

EXERCISES

1. Write a program to calculate and store the contents of various registers as per the following equations.

$$(R1) = ((R1) \text{ AND } (R4)) \text{ XOR } ((R3) \text{ AND } (R2))$$

$$(R2) = (R1) \text{ AND } (R4)$$

$$(R3) = (R1) \text{ XOR } ((R2) \text{ AND } (R4))$$

$$(R4) = (R4) \text{ OR } (R1)$$

2. Considering the original values of R1, R2, R3 and R4 to be the same as in Example 13.7, execute the instructions to find the final values in these registers.

13.10 SHIFT GROUP

This group contains the operations on byte, word or double word. The operations supported are shift left by the number of bits specified as one of the operands, shift right by the number of bits specified in the operand and shift right arithmetic by the number of bits specified in the operand. The addressing modes in all these instructions are

- Count—Immediate or Register
- Operand—Register

The operation can be performed on byte, word or double word operand specified as register. The count (number of bits) may be specified as the immediate value or as the register contents.

The count may be 0 to 8 (for byte operand), 0 to 15 for word operand and 0 to 31 (for double word operand). When specified as register, the address of the register should be between 24 to 255 since 0–23 are the special function registers. The shift operations on special function registers are illegal. Following types of shift operations are possible in the 8096.

- Left Shift
- Logical Right Shift
- Arithmetic Right Shift

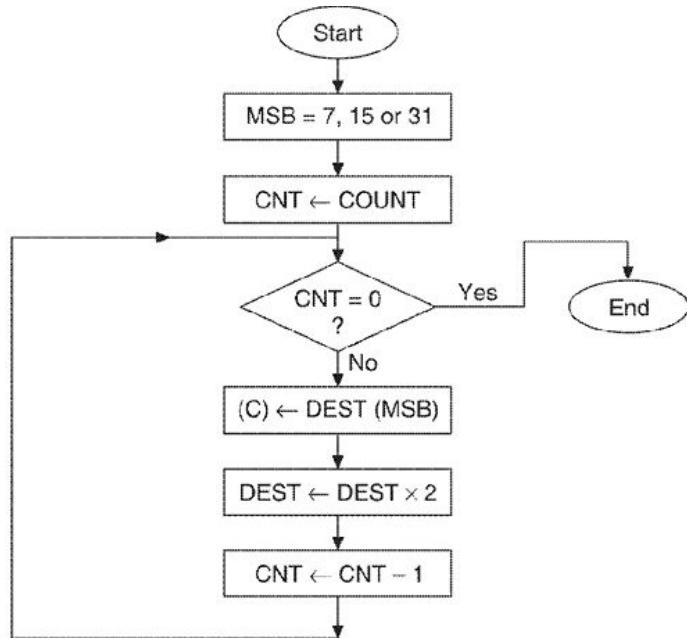
The difference between Logical Right Shift and Arithmatic Right Shift is that, the later preserves the negative sign of the numbers if the original number was negative (i.e. in 2s complement form). Thus, -20 when arithmetic right shifted, would become -10. Let us now deal with the various instructions under these operations.

Left shift

The contents of the operand are shifted left by the specified number of bit positions. The right bits of the result are filled with 0s. The last bit

shifted is saved in the carry flag. The operation logic is explained in Figure 13.12. The flags affected are

Z	N	C	V	VT	ST
✓	?	✓	✓	↑	—



Note—MSB = 7 for byte operand, MSB = 15 for word operand and MSB = 31 for double word operand
 DEST = Byte, Word or Double word register on which shift is performed
 COUNT = Number of bits to be shifted

Figure 13.12 Operation logic of left shift.

Following are the assembly language formats of the instructions.

SHLB—Shift byte left:

SHLB breg, #count

or

SHLB breg, breg

SHL—Shift word left:

SHL wreg, #count

or

SHL wreg, breg

SHLL—Shift double word left:

SHLL Ireg, #count

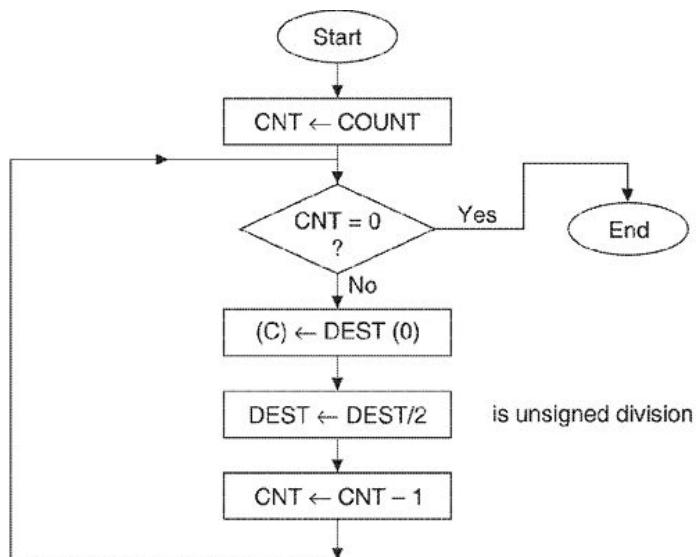
or

SHLL Ireg, breg

Logical right shift

The operation logic is shown in Figure 13.13. The left bits of the result are filled with 0s. The last bit shifted out is saved in the carry. The sticky big flag is cleared at the beginning of the instruction. It is set if at any time during the shift a 1 is shifted first into the carry flag, and a further shift cycle then occurs. The flags affected are

Z	N	C	V	VT	ST
✓	0	✓	0	—	✓



DEST = Byte, Word or Double word register on which shift is performed
 COUNT = Number of bits to be shifted

Figure 13.13 Operation logic of logical right shift.

Following are the instruction formats.

SHRB—Logical right shift byte:

SHRB breg, #count

or

SHRB breg, breg

SHR—Logical right shift word:

SHR wreg, #count

or

SHR wreg, breg

SHRL—Logical right shift double word:

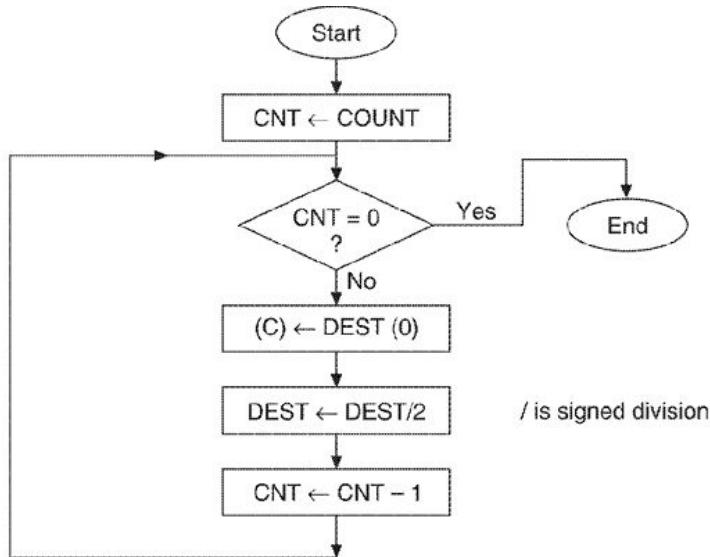
SHRL lreg, #count

or

SHRL lreg, breg

Arithmetic right shift

The operation logic is shown in Figure 13.14. If the original high-order bit (MSB) value is 0, the 0s are shifted in. If MSB is 1, the 1s are shifted in. The last bit shifted out is saved in the carry. The sticky bit flag is cleared at the beginning of the instruction, and set if at any time during the shift, a 1 is shifted first into the carry flag and a further shift cycle occurs.



Note: In signed division if MSB = 1, the ones are shifted otherwise the zeros are shifted.

DEST = Byte, Word or Double word register on which shift is performed
COUNT = Number of bits to be shifted

Figure 13.14 Operation logic of arithmetic right shift.

The flags affected are

Z	N	C	V	VT	ST
✓	✓	✓	0	—	✓

Following are the instruction formats.

SHRAB—Arithmetic right shift byte:

SHRAB breg, #count

or

SHRAB breg, breg

SHRA—Arithmetic right shift word:

SHRA wreg, #count

or

SHRA wreg, breg

SHRAL—Arithmetic right shift double word:

SHRAL lreg, #count

or

SHRAL lreg, breg

EXAMPLE 13.8

Rework Example 13.5 using shift instruction.

Solution:

LD	R1, #4120H
LD	R6, [R1]+ ; (R6) \square M[4120], (R1) \square 4121
SHL	R6, #01H
ADD	R3, R6
LD	R6, [R1]+ ; (R6) \square M[4121], (R1) \square 4122
SHL	R6, #01H
ADD	R3, R6
LD	R6, [R1]+ ; (R6) \square M[4122], (R1) \square 4123
SHL	R6, #01H
ADD	R3, R6
LD	R6, [R1]+ ; (R6) \square M[4123], (R1) \square 4124
SHL	R6, #01H
ADD	R3, R6

Note: R6 is word register.

EXERCISES

1. What will be the contents of R1 after the execution of the following instructions?

LDB	R1, #0FH
LDB	R2, #0AH
SHLB	R1, #04
SHRB	R2, #02
ADDB	R1, R2
ANDB	R1, R2

2. Without using the multiplication and the division instructions, write a program to calculate

$$X = (Y + Z \square 2)/8$$

13.11 BRANCH GROUP

In this group, conditional and unconditional jump operations as well as subroutine call and return operations are supported. The instructions can be put in the following sub-categories.

- Unconditional jumps and subroutine calls.
- Conditional jumps based on different conditions arising out of the comparison between two operands or flags set after an arithmetic operation.
- Conditional jumps based on the status of a particular bit in the register file.

Unconditional jumps and subroutine calls

SJMP—Short jump: The distance from the end of this instruction to the target label is added to the program counter effecting the jump. The offset from the end of this instruction to the target label must be in the range from -1024 to +1023 both inclusive.

SJMP cadd
(PC) □ (PC) + disp (sign extended to 16 bits)
Flags affected—Nil

LJMP—Long jump: The distance from the end of this instruction to the target label is added to the program counter effecting the jump. The operand may be any address in the entire address space.

LJMP cadd
(PC) □ (PC) + disp
Flags affected—Nil

BR (Indirect)—Branch indirect: The execution continues at the address specified in the operand word register.

DEST
BR [wreg]
(PC) □ (DEST)
Flags affected—NIL

SCALL—Short call: The contents of the program counter (the return address) are pushed on to the stack. Then the distance from the end of this instruction to the target label is added to the program counter effecting the call. The offset from the end of this instruction to the target label must be in the range from -1024 to +1023 both inclusive.

SCALL cadd
 $(SP) \square (SP) - 2$
 $((SP)) \square (PC)$
 $(PC) \square (PC) + \text{disp}$ (sign-extended to 16 bits)
 Flags affected—Nil

LCALL—Long call: The contents of the program counter (the return address) are pushed on to the stack. Then the distance from the end of this instruction to the target label is added to the program counter effecting the call. The operand may be any address in the entire address space.

LCALL cadd
 $(SP) \square (SP) - 2$
 $((SP)) \square (PC)$
 $(PC) \square (PC) + \text{disp}$
 Flags affected—Nil

RET—Return from subroutine: The PC is popped off the top of the stack

RET
 $(PC) \square ((SP))$
 $(SP) \square (SP) + 2$
 Flags affected—Nil

Conditional jump based on operands

These include the following:

Mnemonic	Description	Condition
JC	Jump if carry flag is set	$(C) = 1$
JNC	Jump if carry flag is clear	$(C) = 0$
JH	Jump if higher (unsigned)	$(C) = 1 \text{ AND } (Z) = 0$
JNH	Jump if not higher (unsigned)	$(C) = 0 \text{ OR } (Z) = 1$
JE	Jump if equal	$(Z) = 1$
JNE	Jump if not equal	$(Z) = 0$
JV	Jump if overflow flag is set	$(V) = 1$
JNV	Jump if overflow flag is clear	$(V) = 0$
JGE	Jump if signed greater than or equal	$(N) = 0$
JLT	Jump if signed less than	$(N) = 1$
JVT	Jump if overflow trap is set	$(VT) = 1$

JNVT	Jump if overflow trap is clear	(VT) = 0
JGT	Jump if signed greater than	(N) = 0 AND (Z) = 0
JLE	Jump if signed less than or equal	(N) = 1 OR (Z) = 1
JST	Jump if sticky bit is set	(ST) = 1
JNST	Jump if sticky bit is clear	(ST) = 0

Operation: If condition specified is satisfied, then the distance from the end of this instruction to the target label is added to the program counter effecting the jump. The offset from the end of this instruction to the target label must be in the range from -128 to +127. If the condition is not satisfied, then the control passes on to the next sequential instruction.

Jcond. cadd

If cond = True Then

$(PC) \square (PC) + \text{disp}$ (sign extended to 16 bit)

Flags affected—Nil

There is another powerful instruction in conditional jump.

DJNZ—Decrement and jump if not zero: The value of the byte operand is decremented by 1. If the result is not equal to 0, the distance from the end of this instruction to the target label is added to the program counter effecting the jump. The offset from the end of this instruction to the target label must be in the range from -128 to +127. If the result of the decrement is 0, then the control passes to the next sequential instruction.

DJNZ breg, cadd

$(COUNT) \square (COUNT) - 1$

If $(COUNT) <> 0$ Then

$(PC) \square (PC) + \text{disp}$ (sign extended to 16 bits)

Note: $(COUNT) = \text{contents of byte operand}$

Flags affected—Nil

Conditional jump based on bit status

There are two instructions to support the operations—one for Bit Set and the other for Bit Clear condition.

JBC—Jump if bit clear: The specified bit is tested. If it is clear (i.e. 0), then the distance from the end of this instruction to the target label is added to the program counter effecting the jump. The offset from the end of this instruction to the target label must be in the range from -128 to

+127. If the bit is set (i.e. 1), then the control passes to the next sequential instruction.

JBC breg, bitno, cadd
If (specified bit) = 0 Then
(PC) □ (PC) + disp (sign extended to 16 bits)
Flags affected—Nil

JBS—Jump if bit set: The specified bit is tested. If it is set (i.e. 1), the distance from the end of this instruction to the target label is added to the program counter effecting the jump. The offset from the end of this instruction to the target label must be in the range from -128 to +127. If the bit is clear (i.e. 0) the control passes to the next sequential instruction.

JBS breg, bitno, cadd
If (specified bit) = 1 Then
(PC) □ (PC) + disp (sign extended to 16 bits)
Flags affected—Nil

EXAMPLE 13.9

Ten numbers are stored in consecutive memory locations starting from 3000H to 3009H. A number ‘X’ stored in byte register R1 is to be searched. If ‘X’ is present in memory, then the word register R2 will contain its location, otherwise R2 will contain 0000.

Solution:

RSEG AT 20H

R1:	DSB	1
R2:	DSW	1
R3:	DSW	1
R4:	DSB	1
R5:	DSB	1

CSEG AT 2100H

LD	R2, #0000H	
LD	R3, #3000H	
LDB	R4, #0AH	
LOOP:	LDB	R5, [R3]+
CMPB	R5, R1	
JE	LAST	
DJNZ	R4, LOOP	
SJMP	LAST1	

LAST:	LD	R2, R3
LAST1:	NOP	
	END	

EXERCISES

1. Write a program to calculate the factorial of a number stored in memory location 3100H. Store the result in 2 bytes at 3101H and 3102H.
2. Write a program to determine whether the roots of a quadratic equation $AX^2 + BX + C = 0$ will be real or imaginary. A, B and C are stored in memory locations 3101H, 3102H and 3103H respectively.
3. The program in Example 13.9 searches for only the first occurrence of number X. Modify the program to find the number of occurrences of X in memory and store in register R2.

13.12 STACK GROUP

It is considered that memory address and program status word will be stored in stack as well as retrieved from it. The operations supported are PUSH Word, PUSH Flags, POP Words and POP Flags.

PUSH—Push word: The specified operand is pushed onto the stack.

DEST	
PUSH waop	
(SP) □ (SP) – 2	Addressing modes—All
((SP)) □ (DEST)	
Flags affected—Nil	

PUSHF—Push flags: The PSW is pushed on top of the stack, and then set to all 0s. This implies that all the interrupts are disabled. Interrupt calls cannot occur immediately following this instruction.

PUSHF	
(SP) □ (SP) – 2	Addressing modes—Direct
((SP)) □ (PSW)	
(PSW) □ 0	

The flags affected are

Z	N	C	V	VT	ST
0	0	0	0	0	0

POP—Pop word: The word on top of the stack is popped and placed at the destination operand.

DEST

POP waop

(DEST) \square ((SP)) Addressing modes—Direct, Indirect, Indexed

(SP) \square (SP) + 2

Flags affected—Nil

POPF—Pop flags: The word on top of the stack is popped and placed in the PSW. Interrupt calls cannot occur immediately following this instruction.

POPF

(PSW) \square ((SP))

Addressing modes—Direct

(SP) \square (SP) + 2

The flags affected are

Z	N	C	V	VT	ST
✓	✓	✓	✓	✓	✓

13.13 SPECIAL CONTROL GROUP

The operations supported in this group are Clear Bytes, Clear Words, Set and Clear Carry Flags, Clear VT (Overflow Trap Flag), Enable and Disable Interrupt System, Reset Microprocessor System. There are also two instructions for No Operation SKIP: (2 byte instructions) and NOP (1 byte instruction).

CLR—Clear word: The value of the word operand is set to 0.

DEST

CLR wreg

(DEST) \square 0

The flags affected are

Z	N	C	V	VT	ST
1	0	0	0	—	—

CLRB—Clear byte: The value of the byte operand is set to zero.

DEST
CLRB breg
(DEST) \square 0

The flags affected are

Z	N	C	V	VT	ST
1	0	0	0	—	—

SETC—Set carry flag: The carry flag is set.

SETC
(C) \square 1

The flags affected are

Z	N	C	V	VT	ST
—	—	1	—	—	—

CLRC—Clear carry flag: The value of the carry flag is set to zero.

CLRC
(C) \square 0

The flags affected are

Z	N	C	V	VT	ST
—	—	0	—	—	—

CLRVT—Clear overflow trap: The overflow trap flag is reset.

CLRVT
(VT) \square 0

The flags affected are

Z	N	C	V	VT	ST
—	—	—	—	0	—

RST—Reset system: The PSW is initialized to 0, and the PC is initialized to 2080H. The I/O registers are set to their initial value. Executing this instruction will cause a pulse to appear on the reset pin of the 8096.

RST

(PSW) \square 0

(PC) \square 2080H

The flags affected are

Z	N	C	V	VT	ST
0	0	0	0	0	0

DI—Disable interrupts: Interrupts are disabled. Interrupt calls will not occur after this instruction.

D1

Interrupt Enable (PSW. 9) \square 0

Flags affected—Nil

EI—Enable interrupts: Interrupts are enabled following the execution of the next statement. Interrupt calls cannot occur immediately following this instruction.

EI

Interrupt Enable (PSW. 9) \square 1

Flags affected—Nil

NOP—No operation: Nothing is done. Control passes to the next sequential instruction.

NOP

Flags affected—Nil

SKIP—Two byte no-operation: Nothing is done. This is actually a 2-byte NOP where the second byte can be any value, and is simply ignored. The control passes to the next sequential instruction.

SKIP breg

Flags affected—Nil

13.14 OTHER INSTRUCTIONS

There are two other instructions which may help in arithmetic operations.

EXT—Sign extend integer into long-integer: The low-order word of the operand is sign extended throughout the high-order word of the operand.

DEST

EXT lreg

If (low word DEST) < 8000H, Then
 (high word DEST) □ 0
 Else
 (high word DEST) □ FFFFH

The flags affected are

Z	N	C	V	VT	ST
✓	✓	0	0	—	—

EXTB—Sign extend short integer into integer: The low-order byte of the operand is sign extended throughout the high-order byte of the operand.

DEST
 EXTB wreg
 If (low byte DEST) < 80H, Then
 (high byte DEST) □ 0
 Else
 (high byte DEST) □ FFH

The flags affected are

Z	N	C	V	VT	ST
✓	✓	0	0	—	—

NORML—Normalize long integer: The LONG-INTEGER operand is normalized, i.e. it is shifted to the left until its most significant bit is 1. If the most significant bit is still 0 after 31 shifts, the process stops and the 0 flag is set. The number of shifts actually performed is stored in the second operand.

NORML lreg, breg
 (COUNT) □ 0
 Do while (MSB(DEST) = 0) AND ((COUNT) < 31)
 (DEST) □ (DEST) □ 2
 (COUNT) □ (COUNT) + 1
 End_while

Note: COUNT = breg, DEST = Ireg

The flags affected are

Z	N	C	V	VT	ST
✓	?	0	—	—	—

13.15 CONCLUSION

In the present chapter, we dealt with the 8096 addressing modes, instruction set as well as software development. The power of this 16-bit microcontroller becomes evident through these instructions. We shall further experience this while designing systems using the 8096.

EXERCISES

1. Ten numbers are stored in memory locations 3000H to 3009H. Develop a flow chart and write an assembly program to find the minimum number out of the numbers stored and store the minimum number in location 300AH. Use Indirect addressing.
 2. Using Indexed addressing in Exercise 1 above write the program for finding the maximum number.
 3. An industrialist decides that the 1/3 portion of net yearly profit of the company will be kept in bank for emergency. The remaining amount will be divided into five parts. Three parts will be given to his sons. One part will be kept by him and the remaining part will be equally distributed among 192 employees as bonus. If the net yearly profit is stored in the memory location 3000H, write a program to calculate the amount earned by each employee as bonus.
 4. Twenty numbers are stored in memory starting from location 4000H. Write a program to sort them in ascending order.
 5. Modify the above program to sort the numbers in descending order.
 6. There is a safe whose operation is controlled by the 8096 microcontroller in the following manner.
 - (a) The safe has a secret pre-programmed 4-digit code. The user has to enter the code through a set of four thumbwheel switches. Only when the code matches, the safe will open.
 - (b) If the wrong code is entered, it immediately sounds an audio alarm.
- Use the port bits for input code entry and output alarm, and

output to open the safe. Develop the 8096 program for this application.

7. Develop the time of the day clock using the software counters of the 8096. Use time of the day clock in solution to Exercise 6 above to store the time when the code was entered and the output was ‘alarm’ or the ‘safe open’.

FURTHER READING

16 Bit Embedded Controller Handbook, Intel Corporation, Santa Clara.

Douglas, V. Hall, *Microprocessors and Interfacing—Programming and Hardware*, Tata McGraw-Hill, 1999.

Embedded Controllers and Processors Vol. I and Vol. II, Intel Corporation, Santa Clara.

14

THE 8096 MICROCONTROLLER BASED SYSTEM DESIGN CASE STUDIES

14.1 INTRODUCTION

So far we have dealt with the architecture of the 8096 microcontroller along with its interfacing and software. This equips us to design the 8096 microcontroller-based systems, to solve various problems. In this chapter, we will take up two case studies of applications which can be solved using the 8096. These applications will not only demonstrate the power of the 8096, but also lay a base on which you may plan and implement many more applications. While describing the case studies, the relevant process details and fields have been also described in brief so that the application engineer may have a fair knowledge of the process and the field for which the system is being designed. The reader may like to refer to the books and papers mentioned at the end of this chapter, if more in-depth knowledge is required.

14.2 CASE STUDY 1—NUMERICAL CONTROL MACHINE

The numerical control machines are quite popular in mechanical industries where different parts of machines are manufactured. Originally these machines were used to control the machine tools by means of the numbers punched on paper tape. These numbers dictated the movement of the machine tools to produce simple parts such as nuts, bolts or even complex parts of aircraft. The numbers punched on paper tapes were known as part program and the machine as Numerical Control machine or NC machine in short. The earlier machines therefore had a paper tape reader. The modern NC machines are quite different as they are microprocessor controlled. Also high-level part programming languages have been developed and are now commonly used.

The concept of numerical control has been used in different forms by craftsmen since 1650. However, numerical control in the present form did not evolve until the middle of the last century. In 1952, Servomechanism Laboratory of MIT demonstrated the first prototype of a numerically-controlled milling machine. In 1953, MIT made the knowhow available to all interested companies at no cost. By

1960, 5% of the machine tools on display at the National Machine Tool Show were equipped with numerical control and by the 1965 show, most of the manufacturers exhibited some numerical control equipment.

14.3 NC MACHINE CLASSIFICATION

There are different classes of NC machines. Their broad categorization is as follows:

Point-to-point machines

A schematic drawing of a point-to-point NC machine is shown in Figure 14.1. The table serves to position a work piece under the tool. It then remains stationary while some vertical machining operation (drilling, tapping etc.) is performed by the tool. Almost all the machines in this category have the ability to position their tables in $X-Y$ plane with limited ability to operate in the Z direction.

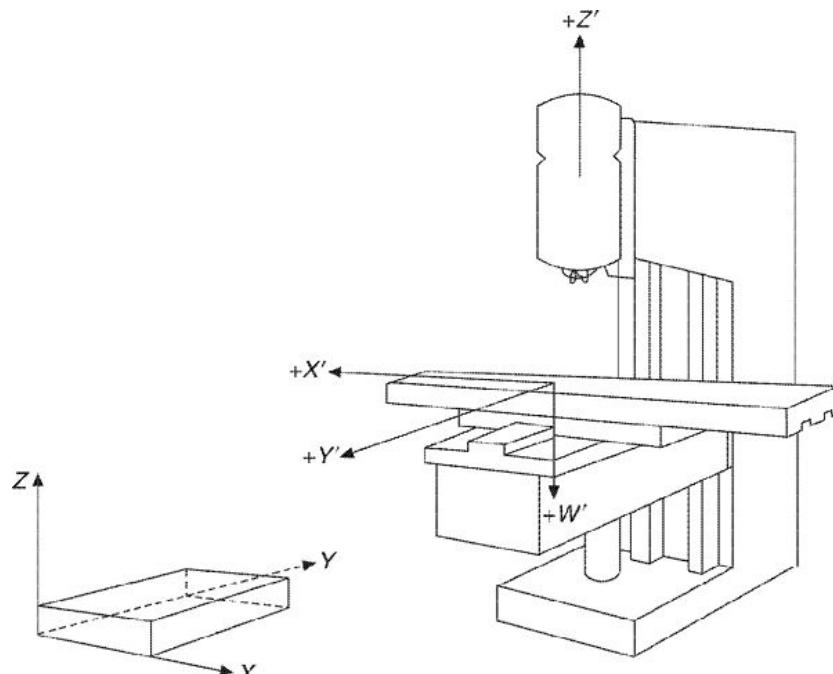


Figure 14.1 Vertical knee mill—a point-to-point NC machine.

Continuous-path machines

Figure 14.2 shows a sophisticated continuous-path NC machine that is capable of cutting virtually any three-dimensional shape by carefully controlling just five axes. In this case the table can be moved in $X-Y$ plane and can also be rotated. The spindle is varied in Z direction and it can be rotated or tilted as well. This type of machine is called the five-axis machine because it has three coordinate motions plus two rotational degrees of freedom.

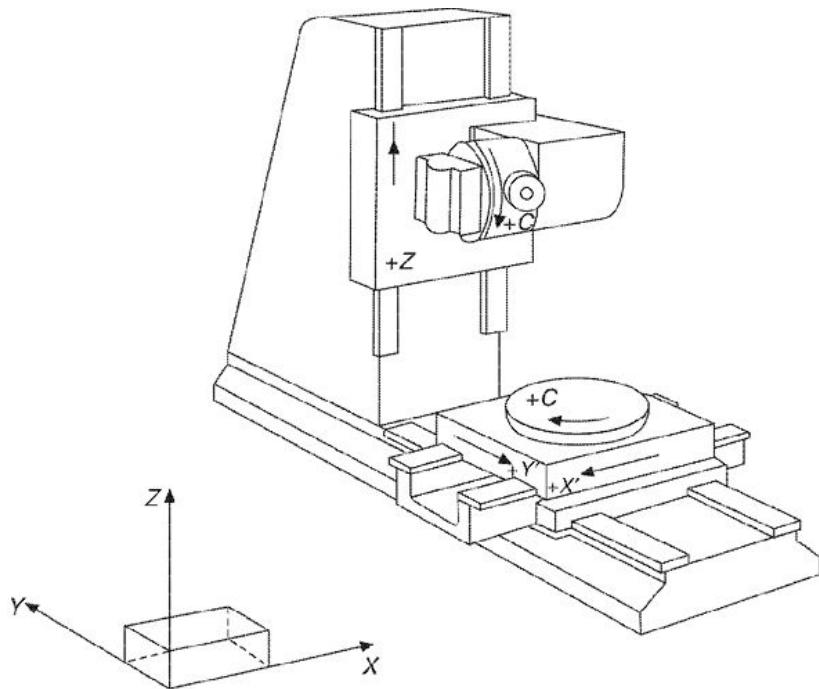


Figure 14.2 A five-axis continuous-path NC machine.

Straight line cutting machines

These are intermediates between point-to-point and continuous-path machines. These machines are used for positioning operations such as drilling. They also have continuous-path capabilities since they are able to machine along straight lines. If these machines are properly programmed they can appropriate the performance of more complex 2-axis contouring machines, for example, they can machine a circular arc by breaking it up into a series of very short line steps. Generally, these systems have only limited third axis motions and are therefore restricted to cutting and positioning in the $X-Y$ plane. In most cases, their straight line cutting capabilities are limited to movements parallel to X - or Y -axis or at an angle of 45° to these axes.

14.4 NUMERICAL CONTROL MACHINE—BASIC DESIGN

The schematic information flow in a basic NC system is shown in Figure 14.3. The sequence of basic manufacturing, engineering and production steps are as follows.

1. The part programmer (the person responsible for the design of the component/part on the NC machine) must first visualize the final product in three dimensions from the blueprint.
2. The second step is to decide the manufacturing procedures, and the type of tools required for the production.
3. The third step is to decide the shape of the part mathematically, involving various techniques—using basic algebra, calculus, and analytical geometry

etc.

4. Having achieved this, the part programmer writes a program to describe the tool motion and management operations required to be performed. This program may be in a high level language. However, it must be translated through a compiler in the form of the numerical coding that can be read and interpreted to control the NC system.
5. The output of the compiler is in the form of numerical codes. The output of the compiler is interpreted (decoded) by the logic of Data Interpretation Section, which converts these codes to another form.
6. A digital-to-analog converter transforms the numerical codes into a series of voltage pulses that describe motions in electrical form. It also sends auxiliary signals that govern a variety of machine tool functions not associated with the table motion, such as on-off commands, coolant flow commands, spindle speed commands, etc. The output of D/A converter controls the servomotor; thus the output shaft is turned by a given amount, to achieve the desired movement in the table.

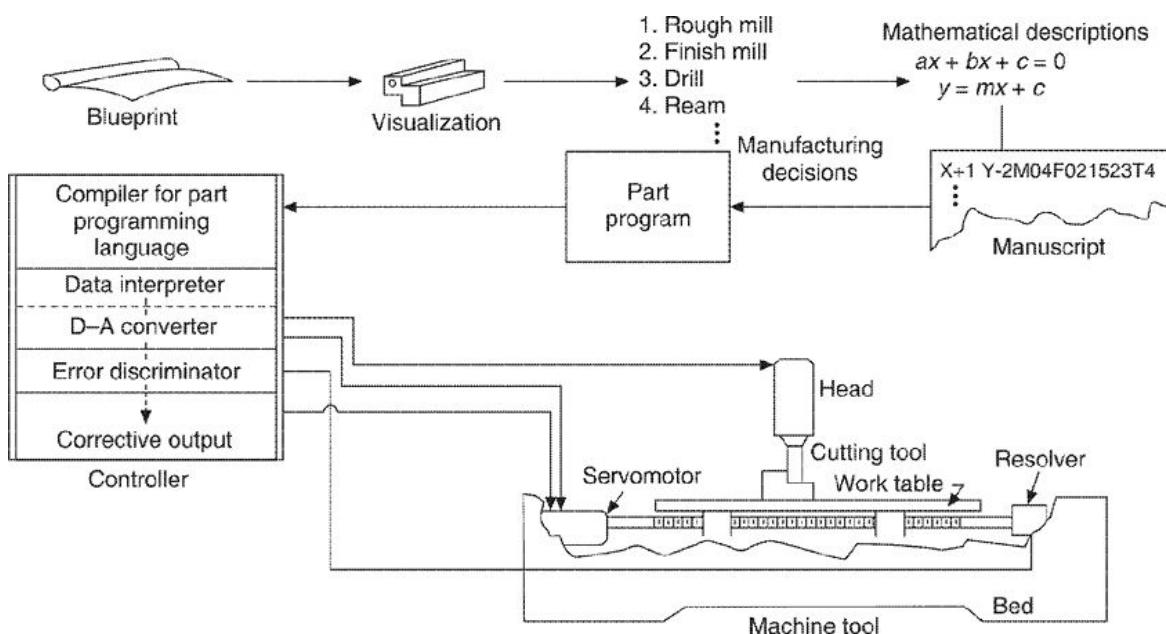


Figure 14.3 Schematic information flow in a basic NC system.

It should be noted that there is no scope for the operator's intuition in NC systems, i.e. there is no chance for the part programmer to pick machinery parameters by feel. All machine tool functions must be precisely determined and coded, so that the system would operate correctly. Even the operations of starting the equipment and stopping after the final cut is made, cannot be neglected. Commands governing table positioning must even be more rigorously planned and programmed.

The motor shaft is mechanically connected to a ball nut lead screw, which converts the rotary motion of the servo to the linear table motion. At the other end of the lead screw, there is a resolver. This device is used to sense the actual motion of the machine tool table and to feed this information back to the controller.

The resolver senses the number of revolutions made by the lead screw and in this way, recognizes the location of the table. This information is converted into another series of voltage pulses which return to the controller. In the controller's Error Discrimination section the feedback pulses are compared with the original command signals. As long as there is discrepancy between the two, a correction signal is produced that is used to drive the servomotor.

In this manner, the machine tool can be used to position at given points while a machining operation is carried out. This is called point-to-point work. However, the machine tool can be made to follow a given path as the manufacturing operations are carried out. This is called the continuous-path operation and is generally associated with the simultaneous motion of two or more axes.

14.5 THE 8096-BASED CONTROL SYSTEM FOR NC MACHINES

Having discussed the NC machines in brief, let us now take up the task of designing the 8096-based control system, for a point-to-point NC machine shown in Figure 14.4.

The NC machine has to be interfaced with two motors, one to rotate the lead screw and the other for the rotation of the spindle. A position transducer must also be incorporated as a resolving transducer.

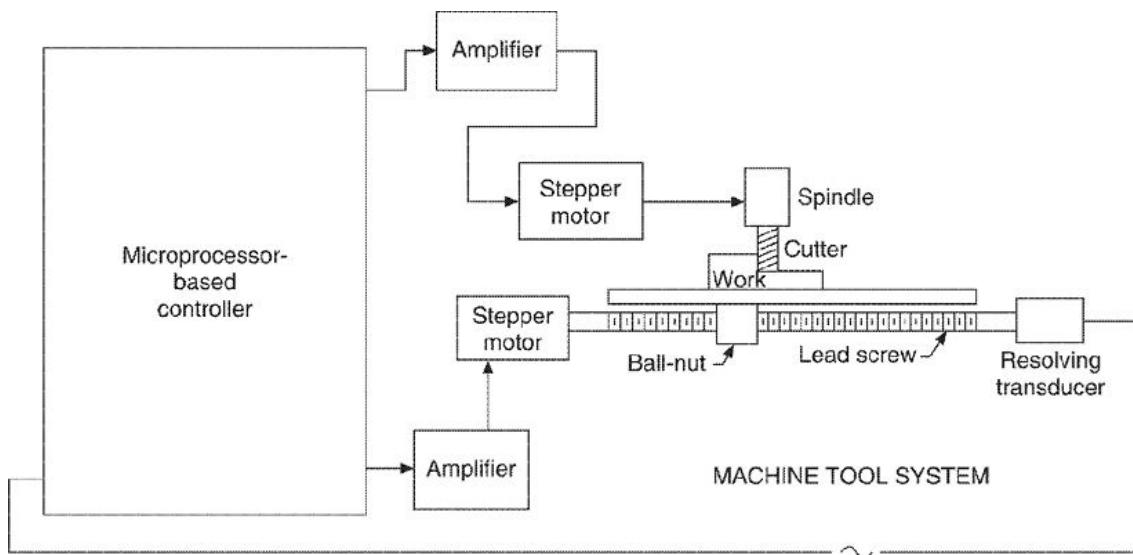


Figure 14.4 Microprocessor-based NC machine controller.

Position transducer

The position transducer can be a simple LVDT (Linear Variable Differential Transformer), a capacitance gauge, or even a fibre optic position transducer. Figure 14.5 shows a simple LVDT. It is basically a transformer with one primary and two secondary coils with a movable core between them. The secondary coils are identical and positioned symmetrically with respect to the primary coil. The secondary coils are connected in phase opposition.

When the core is at the centre, the voltages V_1 and V_2 in both the secondary coils are equal. Thus, voltage $V_o = (V_1 - V_2)$ is zero when the displacement of the core from its central position is zero. When the core is displaced from its central position, V_1 and V_2 have different values due to the symmetry of core location with respect to the secondary coils. This causes a finite nonzero output voltage at V_o . The output of the transducer is linear with the displacement of the core over a wide range and is available in the range of 0 to 5 V which can be directly converted to digital using ADC.

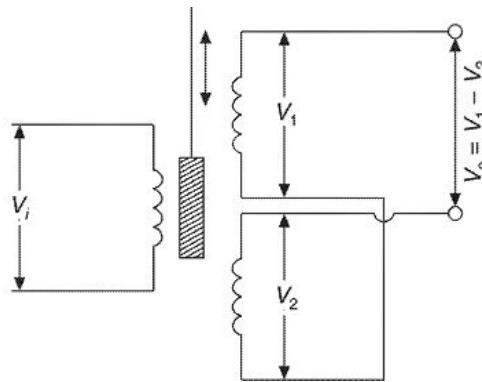


Figure 14.5 Linear variable differential transformer.

Stepper motor

In order to carry out any movement, the NC machines use motors. The motors may be dc motors, ac motors or even universal motors. The motors need to be interfaced to the microprocessor which will control them. Stepper motors are widely used with the microprocessor systems since they can be easily interfaced with them.

The stepper motor, as the name indicates, moves in steps. When the stepper motor drive circuitry receives a step pulse, it drives the motor through a precise angle (step) and then stops until the next pulse is received. Consequently, provided that the maximum permissible load is not exceeded, the total angular displacement of the shaft is equal to the step angle multiplied by the number of the step pulses received. This relation is further simplified, as the shaft position is directly proportional to the number of step pulses supplied, since the step angle for any particular motor is fixed.

The interfacing of the stepper motor requires a circuit which can generate the step pulses at the desired rate and provide the direction signal. It also requires a power amplifier to amplify these low voltage/power signals to the power required by the motor phases. Thus, a microprocessor-based stepper motor interface design can be implemented very easily by using just an output port and a simple software, and a power amplifier circuit.

Let us assume that we have a stepper motor with the following characteristics.

1. Four-phase motor, one phase to be activated at a time.

2. Clockwise direction if the phase activation is in the sequence A, B, C, D,
A and anticlockwise if the phase activation sequence is D, C, B, A, D.
3. Voltage at the common point is 24 V, current per phase is 1.8 A and the
step angle is 1.8° .

The block diagram of the microprocessor interface with the stepper motor is shown in Figure 14.6.

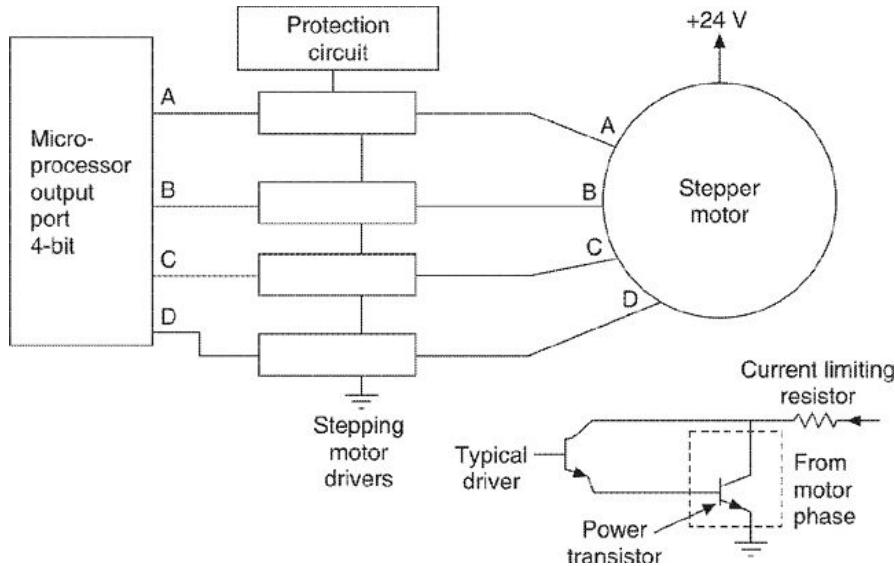


Figure 14.6 Microprocessor interface with stepper motor.

The stepper motor driver circuit will have to be designed specially for a particular motor. A driving transistor (or transistor pair/pairs) capable of providing sufficient gain and desired source/sink current at the desired voltage should be selected. It becomes necessary to have such a circuit, particularly since a motor of high current type needs to have some protection circuit. This circuit is meant to safeguard the motor and to check that only the desired number of phases are 'ON' and the current passing through the stepper motor coils is not dangerously high (otherwise the motor may burn). The circuit may give a feedback to the microprocessor.

Now, the remaining tasks will be done by the microprocessor. It will generate the phase activation signals (on A, B, C, and D lines) on the output port and will wait for the required phase activation time and then, depending upon the direction of rotation, it will activate the corresponding phases. The microprocessor can easily wait for any amount of time starting from a few microseconds to even hours. So, it is very easy to control the speed of the motor. Signals required to move the motor in clockwise direction are shown in Figure 14.7.

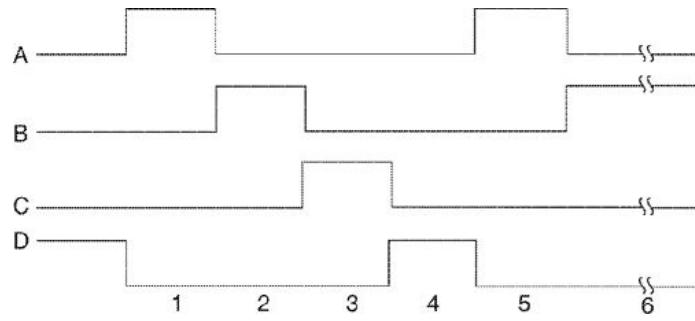


Figure 14.7 Signals for stepper motor movement.

A stepper motor with 1.8° step angle, four-phase, permanent magnet type has been interfaced to the microprocessor as shown in Figure 14.8. A 4-bit latch (IC 7475) is connected to the data bus of the microprocessor and its output is amplified through transistor 2N3055. This motor can move from low speed to up to 30,000 steps per minute, i.e. 150 rpm. It is also possible to move the motor by one-half step. Figure 14.9 shows the phase activation sequence for the full step (1.8°).

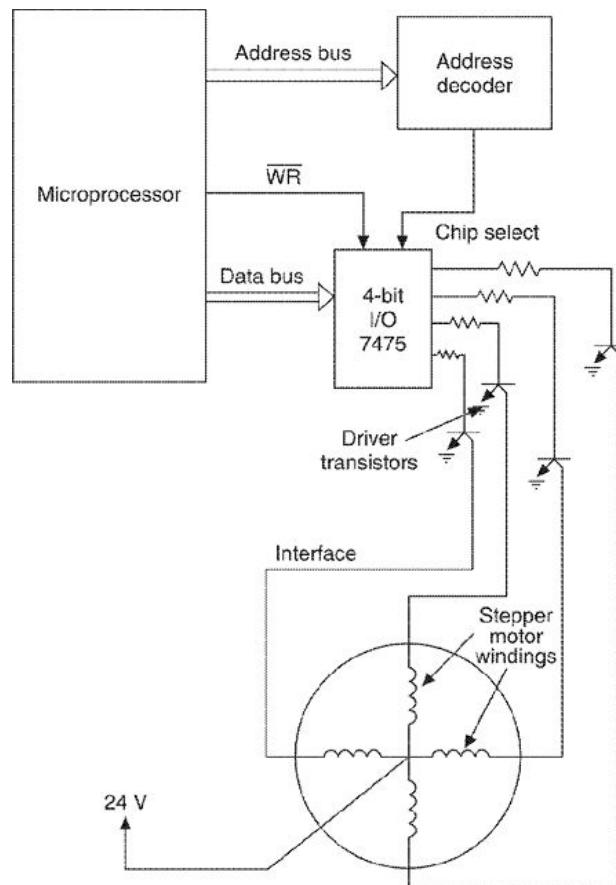


Figure 14.8 Stepper motor control by microprocessor—block diagram.

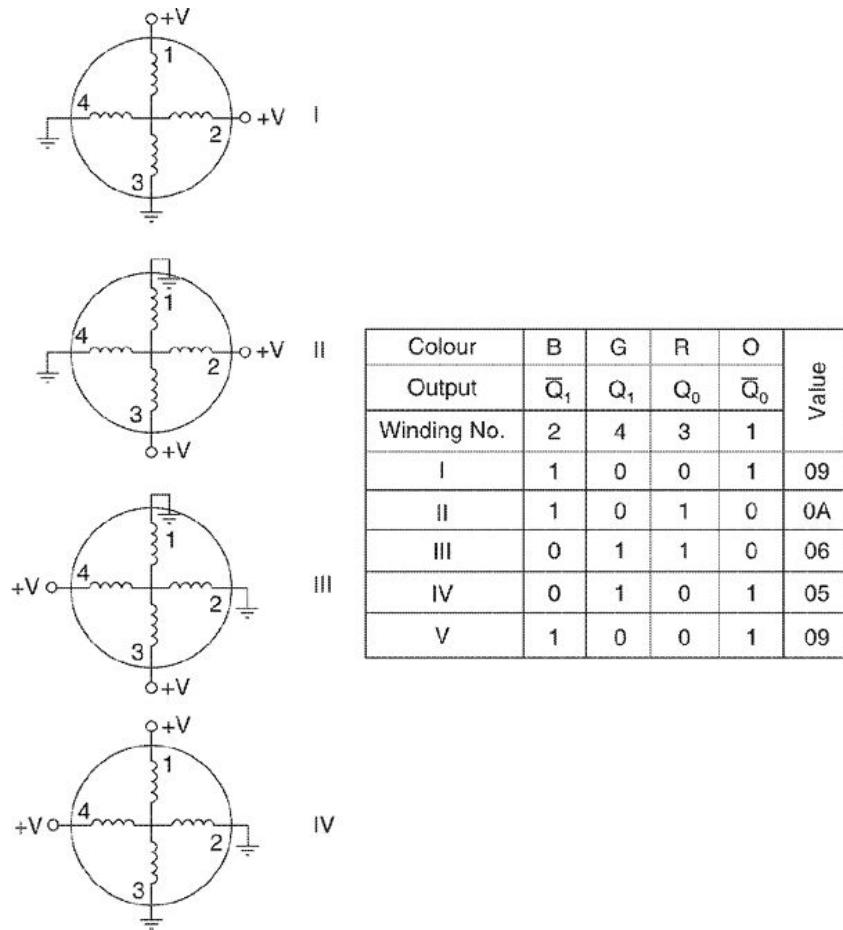


Figure 14.9 Phase activation sequence of stepper motor.

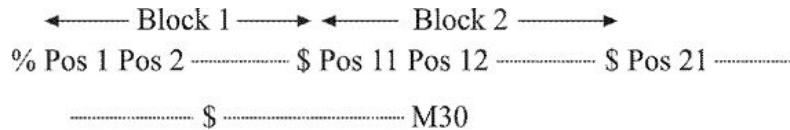
14.5.1 Interfacing NC Machine to the 8096

In terms of interface of an NC machine to the 8096:

- Position transducer output may be connected to the analog input channel.
- There will be two stepper motors. Each stepper motor will require four output lines at the 8096. One of the ports may be assigned for this purpose and its lines may be connected to the two stepper motors. In this case, we are considering that only one motor will be working at a time, i.e. first the positioning and then the cutting will be performed.
- In case it is a continuous path machine, then both the stepper motors will be working simultaneously. In such a case, two ports will be used.

The desired wait time between the two phase activations can be implemented through a software timer.

The part program is stored in memory. The 8096-based controller should execute the part program for performing any job. The part program will consist of different blocks of positions on a metal sheet where some task like boring/drilling/cutting with the same tool is to be performed. The part program, thus, will look like



where

% is the start of the part program

\$ is the end of block.

M30 is the end of part program. It is the miscellaneous instruction to switch off the machine.

When one block of instructions has been executed, the machine should stop and the ‘End of Block’ LED should glow to facilitate the operator to change the tool in the spindle, reposition the sheet and restart. Similarly, at the end of the program, the ‘End of Program’ LED should glow to facilitate the operator to switch off the machine.

Figure 14.10 shows the block diagram of a point-to-point NC machine control using the 8096. The analog input points are represented on Port 0, and Port 1 has been used for the interface to stepper motors. However, instead of Port 1, Port 3 or 4 can also be used. The application does not require external memory since on-chip ROM of 8 KB (8395 and 8397 versions of the 8096) should suffice for the program which can be permanently fused in ROM after development. This will free Ports 3 and 4 which are otherwise used for address/data bus.

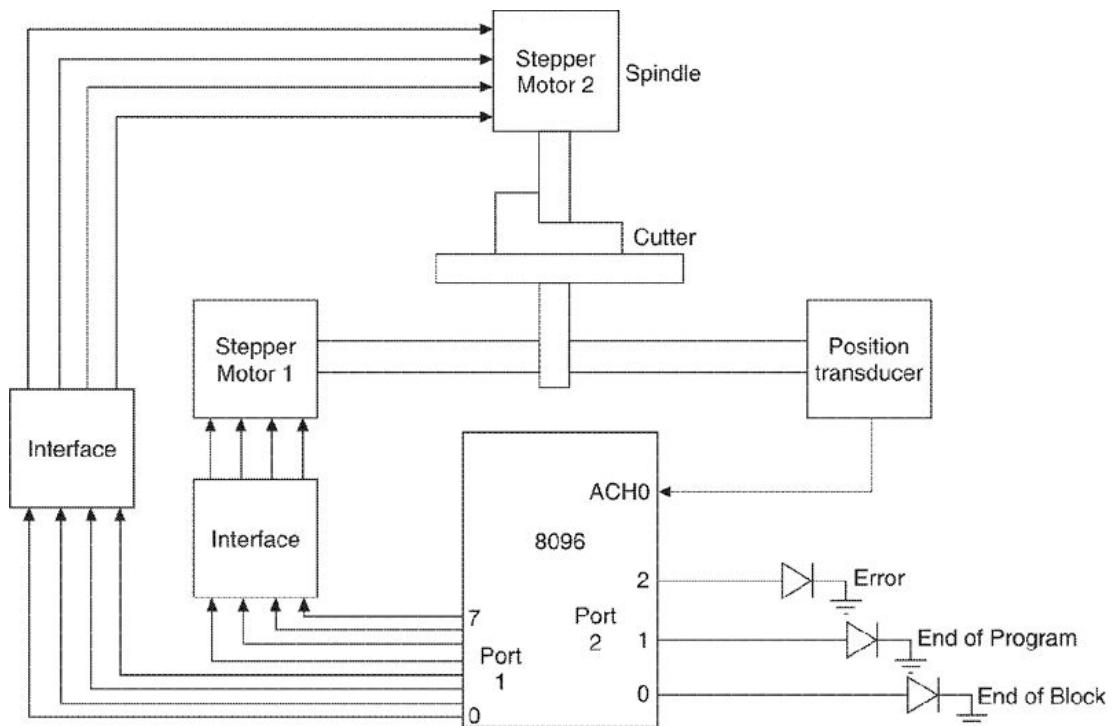


Figure 14.10 Block diagram of the 8096-based controller for point-to-point NC machine.

Three lines of Port 2 (P2.0, P2.1 and P2.2) are used to interface the End of Block LED, the End of Program LED and the Error LED respectively. In case some general purpose programs are required or the 8096 is being used for the

development of an NC machine controller, then Ports 3 and 4 can be used for external memory interfacing.

The system would perform the tasks in the following sequence.

1. Initialization
 2. Read the ‘desired position’ from memory
 3. Read the ‘actual position from transducer
 4. If (desired position – actual position) = 0, then execute step 6 else next step
 5. Move Stepper Motor 1 by one step
 Go to Step 3
 6. Rotate Stepper Motor 2 for fixed time duration
 7. Read the next desired position from memory
 8. Is desired position = \$ (End of Block)?
 If yes then go to Step 9
 else. Is the desired position = M30 (End of Program) ?
 If yes then go to Step 10
 else go to Step 3
 9. Glow End of Block LED
- Stop
10. Glow End of Program LED
- Stop
- End

14.5.2 Software

We shall now develop the software for one block of part program execution of the 8096-controlled NC machine. It is assumed that part program is stored in memory starting from location 2100H.

RSEG AT 20H

P1:	EQU	0FH
P2:	EQU	10H
AD_COMMAND:	EQU	02H
AD_RESULT:	EQU	02H
IMR:	EQU	08H
SP:	EQU	18H
R1:	DSB	1
R5:	DSW	1
R7:	DSW	1
DES_POS:	DSW	1
ACT_POS:	DSW	1
TEMP:	DSW	1
;		
; Store the address of ISR for A/D Completion in interrupt vector location, i.e. 2002H.		
;		
CSEG AT 2002H		

DCW AD_COMPL_INT
 CSEG AT 2100H
 PART_PROG: DSW 0400H; (=1024 Decimal)
 CSEG AT 2800H
 ; Initialization
 ; Since ADC is being used, the SFR AD_RESULT and AD_COMMAND will
 be used.
 ; Mask all interrupts except A/D Completion.
 ; Note IMR bit = 1 – Interrupt enabled
 ; IMR bit = 0 – Interrupt disabled
 ; IMR = 00000010 = 02H
 LDB IMR, #02H
 EI
 ; Define stack area.
 LD SP, #0100H
 ; Initialize LED indications.
 LDB P2, #00H
 ;
 ; Read the first word using indirect addressing and check for % start sign.
 ;
 LD R5, #2100H
 LD DES_POS, (R5)+
 ; Compare with 25H, i.e. ASCII code of %.
 CMP DES_POS, #25H
 JE START
 ; Glow Error indication LED and stop.
 STB P2, #04H
 SJMP B-END
 ; Read the first desired position data.
 START: LD DES_POS, (R5) +
 ; Read the actual position through ADC. Issue the ADC command.
 ; A_COMMAND SFR location -02H

7	6	5	4	3	2	1	0
X	X	X	X	GO			

↔ Channel no.
→ Start Conversion

; For ACH0, AD_COMMAND = 08H
 NEXT_STP: LDB AD_COMMAND, #08H
 ; Wait for A/D Completion ISR.

```

WAIT:           SJMP          WAIT
; After completion of A/D Completion ISR, ACT_POS contains the actual
position.
; Compare DES_POS and ACT_POS.
    CMP          DES_POS, ACT_POS
    JGT          KK1; DES_POS > ACT_POS
    JLT          KK2; DES_POS < ACT_POS
;
; DES_POS = ACT_POS. Start Stepper Motor 2 for a fixed time interval.
    SCALL        STMOTOR2
    SJMP         NEXT_POS
;
; Move Stepper Motor 1 by one step in clockwise direction.
KK1:           SCALL        STMOTOR1_CLK
;
; Insert 1 ms delay to allow movement of stepper motor.
    SCALL        WAIT
    SJMP         NEXT_STP
;
; Move Stepper Motor 1 by one step in anticlockwise direction.
KK2:           SCALL        STMOTOR1_ACLK
;
; Insert 1 ms delay to allow movement of stepper motor.
    SCALL        WAIT
    SJMP         NEXT_STP
;
; Read the next desired position.
NEXT_POS:      LD           DES_POS, (R5) +
;
; Compare with $ sign (24H = ASCII code for $) for End of Block.
    CMP          DES_POS, #24H
    JE           END_BLK
    SJMP         NEXT_STP
;
; End of Block. Glow LED indication and stop.
END_BLK:       LDB          P2, #01H
B-END:          SJMP        B-END
;
; Subroutine
Subroutine     STMOTOR1_CLK
;
; Subroutine to move Stepper Motor1 by one step in clockwise direction.
;
STMOTOR1_CLK:   LDB          P1, #90H
                LDB          P1, #A0H
                LDB          P1, #60H

```

LDB	P1, #50H	
RET		
;	Subroutine	WAIT
;		
;	Subroutine to insert 1 ms delay.	
;		
WAIT:	LDB	R1, #37H; (=55 decimal)
WAIT1:	DJNZ	R1, WAIT1
	RET	

Since most of the time jump will be executed by DJNZ instruction, the execution time will be 9 clock cycles, i.e. 18 ms. For delay of 1 ms, the loop must be executed for $1000/18 = 55$ times, i.e. = 37H.

$$33 \square 18 = 990 \text{ ms}$$

LDB execution time = 4 clock cycles = 8 ms

RET execution time = 12 ms

Thus subroutine will cause delay of 1010 ms, i.e. 1.01 ms.

;	Subroutine	STMOTOR1_ACLK
;		
;	Subroutine to move Stepper Motor1 by one step in anticlockwise direction.	
;		
STMOTOT1_ACLK:	LDB	P1, #50H
	LDB	P1, #60H
	LDB	P1, #A0H
	LDB	P1, #90H
	RET	
;	Subroutine	STMOTOR2
;		

; Let us assume that for each operation Stepper Motor 2 must run in clockwise direction

; for 10 seconds. We can create a delay of 10 seconds in the same way as in the case of

; subroutine wait. The program will be similar to STMOTOR1_CLK.

;

STMOTOR2:	LD	R7, #2534H; (= 9524
Decimal)		
CONTINUE:	LDB	P1, #09H
	LDB	P1, #0AH
	LDB	P1, #06H
	LDB	P1, #05H

; Introduce delay of 1 ms to enable stepper motor

; move in one step.

DJNZ	R7, CONTINUE
SCALL	WAIT
RET	

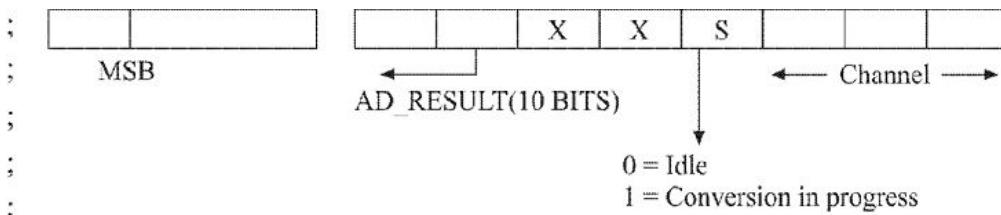
The execution time of loop =

$$\begin{aligned} \text{DJNZ execution time} + 4 \square (\text{execution time of LDB}) + 1 \text{ ms} \\ = 18 \text{ ms} + 4 \square 8 \text{ ms} + 1 \text{ ms} = 1050 \text{ ms} \end{aligned}$$

For running the stepper motor for 10 seconds

$$\text{no. of iterations} = (10 \square 10^6)/1050 = 9524 = 2534H$$

; ISR FOR A/D COMPLETION



CSEG AT 3000H

AD_COMPL_INT:	PUSHF
ST	AD_RESULT, TEMP
SHR	TEMP, #06H
ST	TEMP, ACT_POS
POPF	
EI	
RET	
END	

Thus we have been able to design the 8096-based controller for the point-to-point NC machine.

EXERCISES

1. The case study deals with only one block of the part program. Redesign the application software for a complete part program execution. Incorporate the change in the hardware design, if desired.
2. The delay routine was incorporated in the subroutine for Stepper Motor 2 running for 10 s. Try the same using Timer 1 or Timer 2.

14.6 CASE STUDY 2—AUTOMATION OF WATER SUPPLY FOR A COLONY

We shall now take up the task of designing an automation system for water supply for a residential colony using the 8096 microcontroller.

The present system has been shown in Figure 14.11. There are 2 tanks, an underground tank and an overhead tank. The water from the Municipal Corporation/Water Supply Authority of the city comes to the underground tank

through gravity and later it is pumped to an overhead tank (OHT) using a pump-motor set.

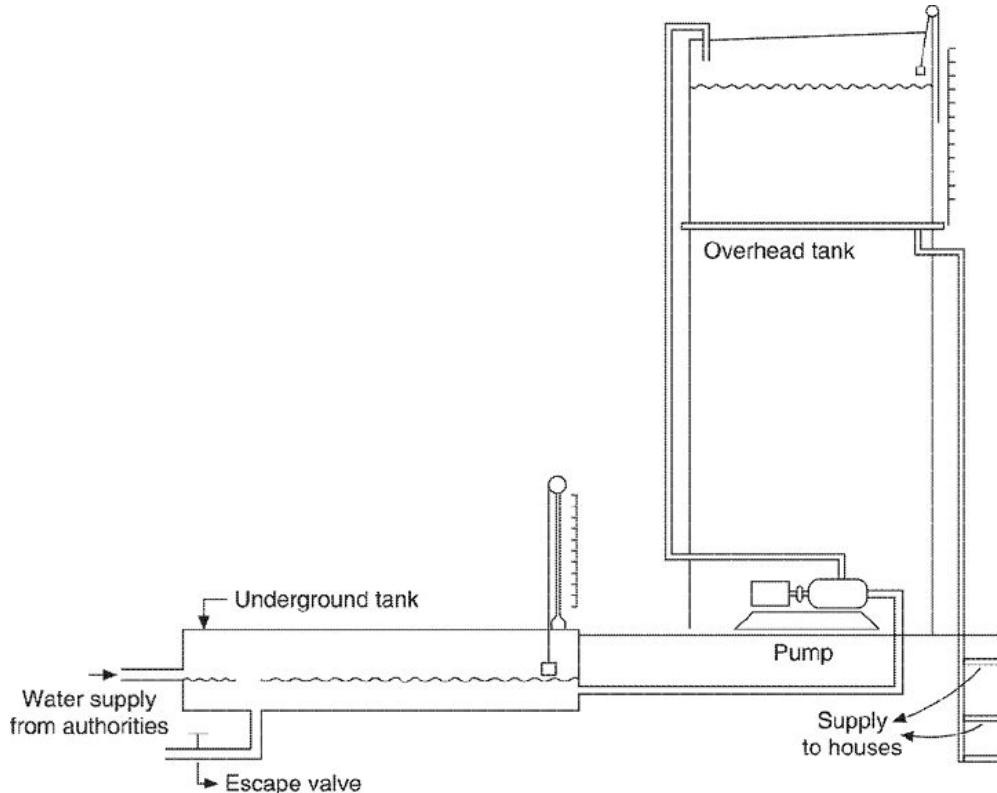


Figure 14.11 Schematic diagram of a water supply distribution system for a colony.

Water is supplied to various residential units from the overhead tank based on the level in the tank. There is an escape valve in the underground tank (UGT) which operates automatically if the underground tank becomes full, a condition which seldom occurs.

In the present system, the levels in the underground tank as well as the overhead tank are measured through a float and pulley sensor and the pump motors are operated manually. Similarly, the distribution valves for water distribution to different residential units are also operated manually. It is an ineffective control prone to human error and leads to wastage of water, shorter life of motors, etc.

It is desired to have a system which can operate automatically without manual intervention/supervision. Following are the desired system features.

The system to be designed should

- operate the water supply system as per the given timetable.
- control the motors to pump the water to OHT based on water present in UGT.
- display the present values of water levels in tanks, status of motors, etc.

The developed system should perform the following functions.

1. Level sensing and control of water in the overhead tank
2. Level sensing of water in the underground tank
3. Start and stop of pump motors for pumping water to the overhead tank from the underground tank
4. Distribution of water to the residential units
5. Display of the system parameters

Level sensing

The present system of level sensing uses floats. A float is a mechanical system, prone to drift with wear and tear, also requiring continuous supervision. Let us first consider the operations being carried out based on the levels of water in the underground and the overhead tanks.

Limits for OHT and UGT

UGT	OHT
Low level = 30 cm	Low level = 60 cm
High level = 3 m	High level = 5.5 m

Operation

UGT Level	OHT Level	Operation
Low	Low	System shutdown
Greater than low	Between low and high	Switch on pump motor
Less than low	Any level	Switch off pump motor

The high level in UGT and the high level in OHT have not been considered as they do not occur in practice. If a high level happens, then the escape valve is opened to supply water to the colony directly from UGT.

For level sensing we can have the type of sensors mentioned in Chapter 8 where the mining problem was discussed in detail. X and Y can tell us the low level and high level in OHT as well as in UGT. Alternatively, ultrasonic sensors may also be used which sense the water level based on the principle of echo received from the bottom of water tank.

Another strategy may be to have pressure sensors installed at the bottom of UGT and OHT and measure the water levels in real time. In case of ultrasonic sensors and pressure sensors, actual water level would be available at the sensor output and the same needs to be digitized and compared with limits to detect the low level and the high level.

Level control

Presently the water is pumped from UGT to OHT through an induction motor-pump set which is operated manually. The automatic operation of the induction motor pump can be attempted through triacs which are nothing but two thyristors connected in the opposite directions. Triacs and thyristors are commercially available and can be readily interfaced to microprocessors.

Also known as silicon controlled rectifier (SCR), a thyristor is basically a three-terminal device like the transistor, but it behaves more like a semiconductor diode. The circuit symbol of the thyristor is shown in Figure 14.12. Either the ac or dc voltage signal may be used as the gate signal, provided the gate voltage is large enough to trigger the thyristor into the ON condition. When triggered, the thyristor allows the current to flow between the anode and the cathode. Thus it can be used to control both ac or dc current in the following manner.

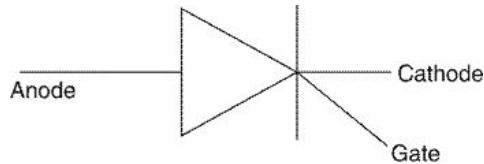


Figure 14.12 Symbol of thyristor.

1. When the input is +ve, and the thyristor is triggered, i.e. the gate voltage is +ve then output = input. If the gate voltage becomes less than the trigger voltage, then the thyristor remains triggered.
2. When the input = 0, the thyristor trigger is reset. If the gate voltage becomes greater than the trigger voltage, the thyristor is triggered.
3. When the Input is -ve, the output = 0, irrespective of whether the thyristor is triggered or not.

Figure 14.13 shows the relationship between the input voltage, the gate voltage and the output voltage of a thyristor.

A triac is basically two thyristors joined back-to-back. Unlike a thyristor, a triac conducts in both the directions and is very well suited for controlling ac voltage and ac powered-devices, like ac motor. The circuit symbol of triac and its characteristic waveforms are shown in Figure 14.14.

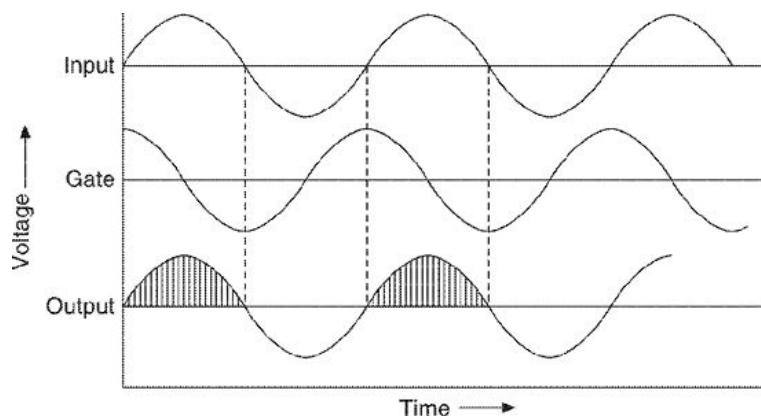


Figure 14.13 Characteristic waveforms of a thyristor.

The triac may act as a power switching device. Solid-state relays based on triacs are popular devices for turning 110 V, 220 V or 440 V ac devices ON or OFF under microprocessor control. However, the control circuit must be electrically isolated from the actual switch, otherwise if accidentally the V_{CC} line

of the microprocessor gets shorted with the ac power line, then the microprocessor and other ICs will receive 110/220/440 V on V_{CC} . This will cause disaster to the whole system.

Figure 14.15 shows the schematic diagram of a solid-state relay using a phototransistor, snubber circuit, trigger circuit, zero crossing detector, and triac.

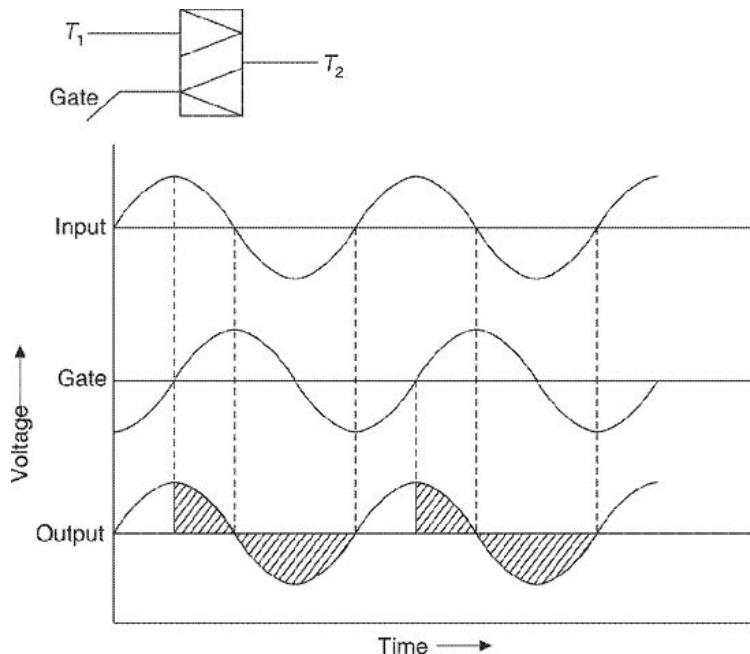


Figure 14.14 Symbol of triac and its characteristic waveforms.

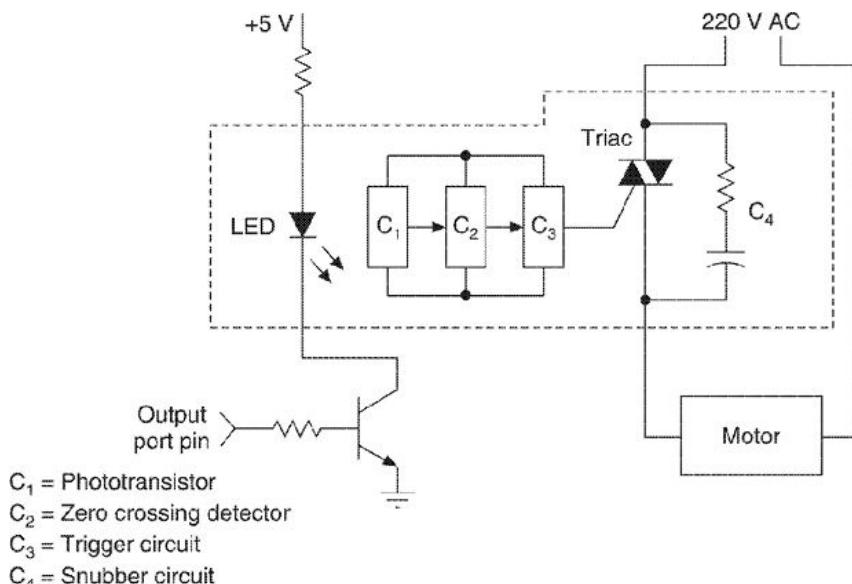


Figure 14.15 Use of triac in motor on/off control.

The input circuit of the solid-state relay is just an LED. A simple *n-p-n* transistor buffer and a current-limiting resistor are all that are needed to interface the relay to a microprocessor output port pin. To turn the relay on, it is required to output a logic '1' on the port pin. This will turn on the transistor and will pull the

required 11 mA through the internal LED. The light from the LED is focused on a phototransistor connected to the actual output-control circuitry.

Since the only connection between the input circuit and the output circuit is a beam of light, there is several thousand volts of isolation between the input circuitry and the output circuitry.

The actual switch in a solid-state relay is a triac. When triggered, this device will conduct on either half of the ac cycle. The zero-voltage detector will make sure that the triac is only triggered when the ac line voltage is very close to one of its zero-voltage crossing points.

When a signal is sent to turn on the relay, the relay will not actually turn on until the next time the ac line voltage crosses zero. This will prevent the triac from turning on at a high-voltage point in the ac cycle, which would produce a burst of EMI. Triacs automatically turn off when the current through them drops below a small value called the holding current; so the triac will automatically turn off at the end of each half-cycle of the ac power.

If the control signal is on, the trigger circuitry will automatically re-trigger the triac for each half-cycle. If a signal is sent to turn off the relay, the triac will always turn on or off at a zero point on the ac voltage. Zero-point switching eliminates most of the EMI that would be caused by switching the triac on at random points in the ac cycle.

A triac will turn off when the current through it drops to near zero. In an inductive circuit, the voltage waveform may be at several tens of volts when the current is at zero. When the triac is conducting, it perhaps has 2 V across it.

When the triac turns off, the voltage across the triac will quickly jump to several tens of volts. This large dV/dT may possibly turn on the triac at a point when it is not desirable. To keep the voltage across the triac from changing too rapidly, an RC snubber circuit is connected across the triac, as shown in Figure 14.15.

The motor to be controlled may be connected as the load. When the logic level at port pin = 1, the motor will be switched on and when it is 0, the motor will be switched off.

14.6.1 Automation of Distribution Valves

In the microcontroller system, the supply timetable would be stored in read only memory. The system clock would provide the time information to the microcontroller. The microcontroller can initiate the water distribution action when the system clock equals the time in the supply timetable. It is necessary to have a battery backup in the system so that the clock provides the time information and does not stop with power failures. Also the system should be capable of becoming operational without manual intervention when the power returns.

It is evident that distribution valves controlled by motors are necessary for the automation of water distribution. There are two options—dc servo motor or stepper motor. In case of dc servo motor, the dc current is supplied to the motor to move the valve in one or the other direction. It is necessary in this case to protect the tightening of valve from more than what is necessary. Thus a feedback

mechanism is required. In case of the stepper motor, the motor moves in response to the pulses applied in step-by-step mode. There is no feedback necessary in this case. The stepper motor and its interface to the microcontroller has already been covered.

The system diagram for water supply distribution is shown in Figure 14.16. It is assumed that for level sensing ultrasonic transducers are installed in both underground and overhead tanks, and the actual water levels in 0 to 5 V range are available at LUGT (for underground tank) and LOHT (for overhead tank). It is evident that many of the facilities of the microcontroller chip have been used in the design of this system. The system would perform the following tasks.

1. Initialization of Timer and ADC SFRs.
2. Scanning the levels in UGT and OHT; status of motors.
3. Converting the analog data to digital (UGT and OHT levels) and storing the data.
4. Processing the data
 - Linearization
 - Conversion into engineering units.
5. Display of OHT level, UGT level, status of motors and distribution timetable.
6. Control of motor pumps based on
 - UGT and OHT levels
 - Status of main and standby motors (operational/inoperational).
7. Control of the distribution valves based on timetable and OHT level.
8. System initialization through keyboard.
9. Self-diagnosis of the system.

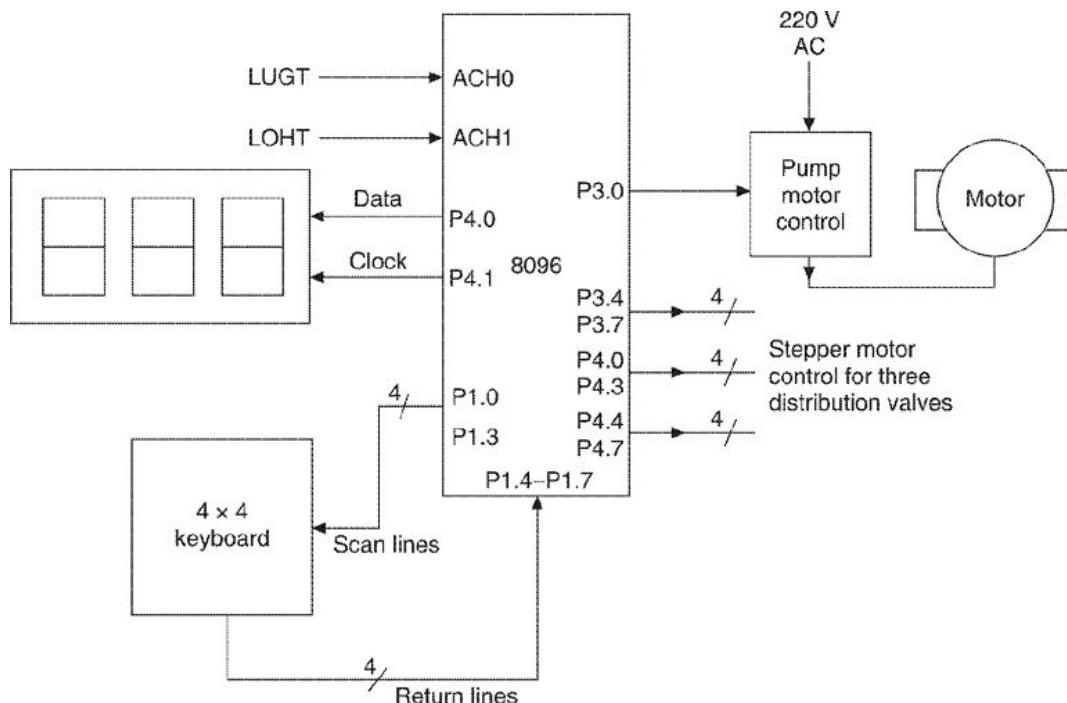


Figure 14.16 The 8096-based control system for water supply distribution.

14.6.2 Software

```
; Initialization
; Since the inbuilt ADC of the 8096 is being used, A/D Completion interrupt
should be
; enabled.
; Similarly since the distribution valves need to be operated based on timetable,
Timer
; Overflow interrupt should be enabled.
; Interrupts enable/masking
; Mask all interrupts except A/D Completion and Timer Overflow.
; A/D Completion—enable
; Timer Overflow—enable
; IMR = 00000011 = 03H
; Note IMR bit = 1 for Interrupt enabled
; IMR bit = 0 for Interrupt disabled
; Considering that we choose to use Timer 1, we need to enable Timer 1
Overflow
```

```
; interrupt.
```

```
; – Timer 1 Overflow—enable
```

```
; IOC1 = 00000100 = 04H
```

```
;
```

```
RSEG AT 20H
```

AD_COMMAND:	EQU	02H
AD_RESULT:	EQU	02H
IMR:	EQU	08H
IOC1:	EQU	16H
SP:	EQU	18H
PM_STATUS:	DSB	1
LOWUGT:	DSW	1
LOWOHT:	DSW	1
HIGHUGT:	DSW	1
HIGHOHT:	DSW	1
LEVEL:	DSW	1
LUGT:	DSW	1
LOHT:	DSW	1
TEMP:	DSW	1
TCOUNT:	DSW	1
TMIN:	DCW	01A4H; (= 420 Decimal)
MINUTE:	DSW	1
THR:	DCW	3CH; (= 60 Decimal)
HOUR:	DSW	1
TDAY:	DCW	18H
SCHHR1-START:	DSW	1

```

SCHMNT1-START:      DSW          1
SCHHR1-STOP:        DSW          1
SCHMNT1-STOP:        DSW          1
SCHHR2-START:        DSW          1
SCHMNT2-START:        DSW          1
SCHHR2-STOP:        DSW          1
SCHMNT2-STOP:        DSW          1

; Store address of ISR for Timer Overflow and A/D Completion interrupts in
interrupt
; vector locations 2000H and 2002H.
;

CSEG AT 2000H
    DCW          TIMER_OV_INT
    DCW          AD_COMPL_INT

CSEG AT 3000H
    LDB          IMR, #03H
    LDB          IOC1, #04H
    EI

;

; Define stack area.
;
    LD           SP, #0100H
;

; Scan UGT, OHT.
;
; AD_COMMAND
;   7   6   5   4   3   2   1   0
;   X   X   X   X   GO   |   |   |
;                         |   |
;                         Channel no.

; Issue A/D command for level in UGT.
ST_LOOP:             LDB          AD_COMMAND, #08H
WT1:                 SJMP         WT1
    ST           LEVEL, LUGT
;

; Issue A/D command for level in OHT.
;
; AD_COMMAND
;   7   6   5   4   3   2   1   0
;   X   X   X   X   GO   |   |   |
;                         |   |
;                         Channel no.

WT2:                 SJMP         WT2
    ST           LEVEL, LOHT
;

; Display parameters.
    SCALL        DISPLAY
;

; Check UGT water level.

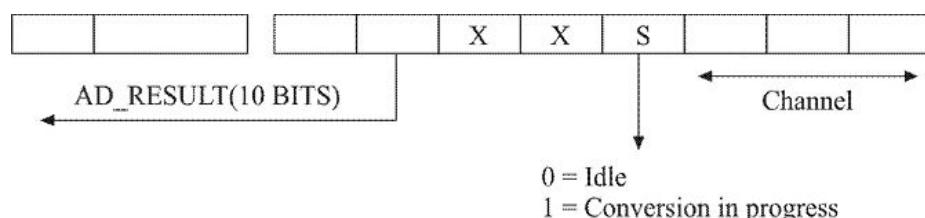
```

```

;
        CMP          LUGT, LOWUGT
; LUGT > LOWUGT.
        JGT          KK1
;
; LUGT < LOWUGT. Stop pump motor if running else no action.
;
; PM_STATUS = 0 – Pump motor not running
; PM_STATUS = 1 – Pump motor running
        CMPB         PM_STATUS, #01H
        JLT          ST_LOOP
; Stop pump motor.
        LDB          P3, #00H
        LDB          PM_STATUS, #00
        SJMP         ST_LOOP
;
; Check OHT water level.
;
KK1:           CMP          LOHT, HIGHOHT
;
; LOHT < HIGHOHT
        JLT          KK2
;
; LOHT > HIGHOHT since OHT level is high, no pumping action desired.
;
        SJMP         ST_LOOP
;
; Check pump status, if running then no action, else start the pump motor.
;
KK2:           CMPB         PM_STATUS, #00H
        JGT          ST_LOOP
; Start the pump motor.
        LDB          P3, #01H
;
; Change the pump motor status.
        LDB          PM_STATUS, #01H
SJMP ST_LOOP

```

ISR For A/D COMPLETION



AD_COMPL_INT:	PUSHF
ST	AD_RESULT, TEMP
SHR	TEMP, #06H
ST	TEMP, LEVEL
POPF	
EI	
RET	

Timer operation

Timer1 receives system clock pulse after every 2 ms.

Thus Timer1 overflow period = 2×2^{16} ms.

$$= 2 \times 10^{-6} \times 1024 \times 64 \text{ second} = 131 \text{ ms}$$

Therefore, in one second the number of overflow pulses will be $1000/131 = 7$.

Thus, the number of overflow pulses in one minute = 420

The real time clock can be designed using the above information.

Let us assume that two variables MINUTE and HOUR represent the actual time of the day at any instant. The user can input these values using the keyboard. An ENTER subroutine may be developed to facilitate the input of the time of the day, distribution time table and low and high limits of water levels in UGT and OHT.

In addition, there are three constants:

TMIN = number of Timer 1 overflow pulses to make 1 minute period = 420

THR = number of minutes to make one hour = 60

TDAY = number of hours to make one day = 24

TCOUNT is the variable which represents the current value of Timer 1 overflow pulses.

- When TCOUNT becomes 420 (value of TMIN), one minute has passed □ MINUTE increases by 1, TCOUNT is reset to 0.
- When MINUTE equals THR, one hour has passed □ HOUR increases by 1, MINUTE is reset to 0.
- When HOUR equals TDAY, one day has passed, HOUR is reset to 0.

The water supply distribution schedule is represented as

Supply	SCHHR1_START	SCHMNT1_START
Schedule1	SCHHR1_STOP	SCHMNT1_STOP
Supply	SCHHR2_START	SCHMNT2_START
Schedule2	SCHHR2_STOP	SCHMNT2_STOP

These constants are stored in memory at predefined locations. The supply is started or stopped when the current time equals the schedule start or the schedule stop time. As an example for supply schedule 1, the water supply will start at SCHHR1_START hour and SCHMNT1_START minute. The supply will stop at SCHHR1_STOP hour and SCHMNT1_STOP minute.

Now, let us develop the software to implement the above logic.

ISR for Timer1 Overflow

```
TIMER_OV_INT:      PUSHF
                    INC           TCOUNT; No. of Timer1 overflow

; Check if TCOUNT = TMIN i.e. 420
        CMP           TCOUNT, TMIN; (TCOUNT – TMIN)
        JLT           KKR

; TCOUNT = TMIN, increase MINUTE, reset TCOUNT
        INC           MINUTE
        LD            TCOUNT, #0000H

; Check if MINUTE has become 60
        CMP           MINUTE, THR; (MINUTE – THR)
        JLT           KKCHK

;
; MINUTE = THR = 60, increase HOUR. Reset MINUTE.
        INC           HOUR
        LD            MINUTE, #0000H

; Check if HOUR has reached 24?
        CMP           HOUR, TDAY; (HOUR – TDAY)
        JLT           KKCHK

; HOUR = TDAY=24. Reset HOUR.
        LD            HOUR, #0000H

;
; Check current time with the distribution start/stop schedule
;
KKCHK:          CMP           HOUR, SCHHR1_START
;
; Not equal to supply start time
        JNE           KKCHK1
;
; Schedule Hour equals current Hour. Check minutes.
;
        CMP           MINUTE, SCHMNT1_START
        JNE           KKCHK1
;
; Start water supply distribution.
        SCALL          DIST_START
        SJMP           KKR

;
; Check for distribution stop schedule.
;
KKCHK1:         CMP           HOUR, SCHHR1_STOP
        JNE           KKCHK2
```

CMP	MINUTE, SCHMNT1_STOP
JNE	KKR
;	
; Stop water supply distribution.	
SCALL	DIST_STOP
;	
; Check for Schedule no. 2.	
;	
KKCCHK2:	—
—	
—	
POPF	
EI	
KKR:	RET
Subroutines	DIST_START and DIST_STOP

Let us assume that the distribution valve will open with clockwise movement of the stepper motor which is controlling the valve. We should also know how many steps will open the valve fully. Based on that subroutine, DIST_START can be easily written to move all the three stepper motors by a specified number of steps. We have already discussed stepper motor control using the 8096 microcontroller and you may easily write the the 8096 program following the same logic.

For stopping the water supply distribution, the stepper motor will have to be moved anticlockwise by the specified number of steps.

Thus we have been able to show how a water supply distribution control system for a colony can be designed using the 8096.

14.7 CONCLUSION

The case studies discussed in this chapter present the power of the 8096 microcontroller. The basic ingredients of the embedded control applications like A-to-D, D-to-A, bit addressing, powerful interrupt facility, built-in timer operation are all present in the 8096 and these are exhibited in these case studies. They make you realize why the 8096 is called the ideal embedded controller. A thorough understanding of these case studies would enable you take up the task of system design of more complex applications.

EXERCISES

1. The program written in Section 14.6.2 is not complete as ENTER, DISPLAY, DIST_START and DIST_STOP subroutines have not been dealt with. The structures of these subroutines have been discussed. Develop the logic, write the 8096 program and integrate it with the program given in the text.

2. In the program developed in Section 14.6.2, the OHT level has not been taken into account before starting the water supply distribution as well as during the water supply distribution. Modify the program to incorporate the following conditions.
 - (a) Initially if LOHT \square LOWOHT—water distribution should not start.
 - (b) When LOHT becomes equal to or less than LOWOHT—water distribution must stop.
 - (c) Both these conditions must be displayed in seven-segment LED display.
3. Test the integrated software in a simulated environment. Assume twenty continuously increasing values of UGT and OHT levels to test the complete software.

FURTHER READING

- 16 Bit Embedded Controller Handbook*, Intel Corporation, Santa Clara.
- Banmaah, H.D., *Trends in control valves and actuator, instruments and control systems*, Oct. 1982.
- Consdine, Douglas, *An Encyclopedia of Instrumentation and Control*, McGraw-Hill Book Company, NY, 1971.
- Dyke, R.M. *Numerical Control*, Englewood Cliffs, New Jersey, 1967.
- Hall, Douglas V., *Microprocessors and Interfacing*, Tata McGraw-Hill, 1999.
- Krishna Kant, *Computer Based Industrial Control*, Prentice-Hall of India, 1997.
- Liptak, B.G., *Instrumentation Engineering Handbook*, Chilton Book Company, Penn. 1985.
- Pusztai, Joseph and Michael Sava, *Computer Numerical Control*, Reston Publishing Co., 1983.
- Simpson, Colin D., *Industrial Electronics*, Prentice Hall, Eaglewood Cliffs, New Jersey, 1996.

Appendix I

INTEL 8085 INSTRUCTION SET SUMMARY

Data Transfer Group

The data transfer instructions move data between registers or between memory and registers.

MOV	Move
MVI	Move Immediate
LDA	Load Accumulator Directly from Memory
STA	Store Accumulator Directly in Memory
LHLD	Load H and L Registers Directly from Memory
SHLD	Store H and L Registers Directly in Memory

Note: An ‘X’ in the name of a data transfer instruction implies that it deals with a register pair (16 bits).

LXI	Load Register Pair with Immediate data
LDAX	Load Accumulator from Address in Register Pair
STAX	Store Accumulator in Address in Register Pair
XCHG	Exchange H and L with D and E
XTHL	Exchange Top of Stack with H and L

Arithmetic Group

The arithmetic instructions add, subtract, increment, or decrement data in registers or memory.

ADD	Add to Accumulator
ADI	Add Immediate Data to Accumulator
ADC	Add to Accumulator using Carry Flag
ACI	Add Immediate Data to Accumulator using Carry
SUB	Subtract from Accumulator
SUI	Subtract Immediate Data from Accumulator
SBB	Subtract from Accumulator using Borrow (Carry) Flag

SBI	Subtract Immediate from Accumulator using Borrow
(Carry) Flag	
INR	Increment specified Byte by one
DCR	Decrement specified Byte by one
INX	Increment Register Pair by one
DCX	Decrement Register Pair by one
DAD	Double Register Add; Add contents of Register Pair to H and L Register Pair

Logical Group

This group performs logical (Boolean) operations on data in registers and memory, and on Condition Flags.

- (i) The logical AND, OR, and Exclusive OR instructions enable you to set specific bits in the accumulator ON or OFF.

ANA	Logical AND with Accumulator
ANI	Logical AND with Accumulator using Immediate Data
ORA	Logical OR with Accumulator
ORI	Logical OR with Accumulator using Immediate Data
XRA	Exclusive OR with Accumulator
XRI	Exclusive OR using Immediate Data

- (ii) The Compare instructions compare the contents of an 8-bit value with the contents of the accumulator.

CMP	Compare
CPI	Compare using Immediate Data

- (iii) The Rotate instructions shift the contents of the accumulator one bit position to the left or right.

RLC	Rotate Accumulator Left
RRC	Rotate Accumulator Right
RAL	Rotate Left through Carry
RAR	Rotate Right through Carry

- (iv) Complement ACC and Carry Flag instructions:

CMA	Complement Accumulator
CMC	Complement Carry Flag
STC	Set Carry Flag

Branch Group

The branching instructions alter the normal sequential program flow, either unconditionally or conditionally.

(i) The Unconditional Branch instructions are as follows:

JMP	Jump
CALL	Call
RET	Return

(ii) The Conditional Branch instructions examine the status of one of the four condition flags to determine whether the specified branch is to be executed. The conditions that may be specified are as follows:

NZ	Not Zero ($Z = 0$)
Z	Zero ($Z = 1$)
NC	No Carry ($C = 0$)
C	Carry ($C = 1$)
PO	Parity Odd ($P = 0$)
PE	Parity Even ($P = 1$)
P	Plus ($S = 0$)
M	Minus ($S = 1$)

Thus, the Conditional Branch instructions are specified as follows:

Jumps	Calls	Returns	
JC	CC	RC	(Carry)
JNC	CNC	RNC	(No Carry)
JZ	CZ	RZ	(Zero)
JNZ	CNZ	RNZ	(Not Zero)
JP	CP	RP	(Plus)
JM	CM	RM	(Minus)
JPE Even)	CPE	RPE	(Parity)
JPO	CPO	RPO	(Parity Odd)

(iii) Two other instructions can affect a branch by replacing the contents of the Program Counter.

PCHL	Move H and L to Program Counter
RST	Special Restart Instruction used with Interrupts

Stack I/O, and Machine Control Instructions

The following instructions affect the Stack and/or Stack Pointer.

PUSH	Push two bytes of Data onto the Stack
POP	Pop two bytes of Data off the Stack
XTHL	Exchange Top of Stack with H and L
SPHL	Move contents of H and L to Stack Pointer

Input/Output Instructions

IN	Initiate Input Operation
OUT	Initiate Output Operation

Machine Control Instructions

EI	Enable Interrupt System
DI	Disable Interrupt System
HLT	Halt
NOP	No Operation

Instruction Set and Opcode (in Hex)

Data Transfer Group

1. MOV r1, r2 States—
4, Flags—None

Inst—MOV	A,A	A,B	A,C	A,D	A,E	A,H	A,L	B,A	B,B	B,C	B,D	B,E
Opcode	7F	78	79	7A	7B	7C	7D	47	40	41	42	43
Inst—MOV	B,H	B,L	C,A	C,B	C,C	C,D	C,E	C,H	C,L	D,A	D,B	D,C
Opcode	44	45	4F	48	49	4A	4B	4C	4D	57	50	51
Inst—MOV	D,D	D,E	D,H	D,L	E,A	E,B	E,C	E,D	E,E	E,H	E,L	H,A
Opcode	52	53	54	55	5F	58	59	5A	5B	5C	5D	67
Inst—MOV	H,B	H,C	H,D	H,E	H,H	H,L	L,A	L,B	L,C	L,D	L,E	L,H
Opcode	60	61	62	63	64	65	6F	68	69	6A	6B	6C
Inst—MOV	L,L											
Opcode	6D											

2. MOV r, M/MOV M, r States—
7, Flags—None

Inst—MOV	A,M	B,M	C,M	D,M	E,M	H,M	L,M
Opcode	7E	46	4E	56	5E	66	6E
Inst—MOV	M,A	M,B	M,C	M,D	M,E	M,H	M,L
Opcode	77	70	71	72	73	74	75

3. MVI r, byte/MVI M, byte States—
7, Flags—None

Inst—MVI	A,byte	B,byte	C,byte	D,byte	E,byte	H,byte	L,byte	M,byte
Opcode	3E	06	0E	16	1E	26	2E	36

4. LXI rp, dble States—
10, Flags—None

Inst—LXI	B,dble	D,dble	H,dble	SP,dble
Opcode	01	11	21	31

5. LDAX rp/STAX rp States—
7 Flags—None

Inst—	LDAX B	LDAX D	STAX B	STAX D
Opcode	0A	1A	02	12

6. LDA addr/STA addr States—
13 Flags—None

Inst—	LDA addr	STA addr
Opcode	3A	32

7. LHLD addr/SHLD addr States—
16 Flags—None

Inst—	LHLD addr	SHLD addr
Opcode	2A	22

8. XCHG Opcode = EB States—
4 Flags—None

Arithmetic Group

1. ADD r/ADC r States—4 Flags—
Z, S, P, CY, AC

Inst—ADD	A	B	C	D	E	H	L
Opcode	87	80	81	82	83	84	85
Inst—ADC	A	B	C	D	E	H	L
Opcode	8F	88	89	8A	8B	8C	8D

2. ADD M/ADC M States—7 Flags—
—Z, S, P, CY, AC

Inst—	ADD M	ADC M
Opcode	86	8E

3. SUB r/SBB r States—4 Flags—
—Z, S, P, CY, AC

Inst—SUB	A	B	C	D	E	H	L
Opcode	97	90	91	92	93	94	95
Inst—SBB	A	B	C	D	E	H	L

Opcode	9F	98	99	9A	9B	9C	9D
--------	----	----	----	----	----	----	----

4. SUB M/SBB M
Z, S, P, CY, AC

States—7

Flags—

Inst—	SUB M	SBB M
Opcode	96	9E

5. ADI byte/ACI byte/SUI byte/SBI byte
S, P, CY, AC

States—7

Flags—Z,
S, P, CY, AC

Inst—	ADI byte	ACI byte	SUI byte	SBI byte
Opcode	C6	CE	D6	DE

6. DAD rp
—CY

States—10

Flags

Inst—DAD	B	D	H	SP
Opcode	09	19	29	39

7. INR r/DCR r
Z, S, P, AC

States—4

Flags—

Inst—INR	A	B	C	D	E	H	L
Opcode	3C	04	0C	14	1C	24	2C
Inst—DCR	A	B	C	D	E	H	L
Opcode	3D	05	0D	15	1D	25	2D

8. INR M/DCR M
—Z, S, P, AC

States—10

Flags

Inst—	INR M	DCR M
Opcode	34	35

9. INX rp/DCX rp
6 Flags—None

States—

Inst—INX	B	D	H	SP
Opcode	03	13	23	33
Inst—DCX	B	D	H	SP
Opcode	0B	1B	2B	3B

10. DAA
Z, S, P, CY, AC

Opcode = 27

States—4

Flags—

Logical Group

1. ANA r/ORAr/XRAr/CMPr States—4 Flags—
Z, S, P, CY, AC

Inst—ANA	A	B	C	D	E	H	L
Opcode	A7	A0	A1	A2	A3	A4	A5
Inst—ORAr	A	B	C	D	E	H	L
Opcode	B7	B0	B1	B2	B3	B4	B5
Inst—XRAr	A	B	C	D	E	H	L
Opcode	AF	A8	A9	AA	AB	AC	AD
Inst—CMPr	A	B	C	D	E	H	L
Opcode	BF	B8	B9	BA	BB	BC	BD

2. ANAM/ORAM/XRAM/CMPM States—7 Flags—
Z, S, P, CY, AC

Inst—	ANAM	ORAM	XRAM	CMPM
Opcode	A6	B6	AE	BE

3. ANI byte/ORI byte/XRI byte/CPI byte States—7 Flags
—Z, S, P, CY, AC

Inst—	ANI byte	ORI byte	XRI byte	CPI byte
Opcode	E6	F6	EE	FE

4. RLC/RRC/RAL/RAR States—7 Flags
—Z, S, P, CY, AC

Inst—	RLC	RRC	RAL	RAR
Opcode	07	0F	17	1F

5. CMA Opcode = 2F States—4 Flags—
None

6. STC Opcode = 37 States—4 Flags—
CY

7. CMC Opcode = 3F States—4 Flags—
CY

Branch Group

Inst	Opcode	States	Flags
1. JMP addr	C3	10	None

2. CALL addr	CD	18	None
3. RET	C9	10	None
4. PCHL	E9	6	None
5. RSTn	States—12		Flags—
None			

Inst—	RST0	RST1	RST2	RST3	RST4	RST5	RST6	RST7
Opcode	C7	CF	D7	DF	E7	EF	F7	FF

6. J. Cond. addr	States—7/10	Flags
—None		

Inst—	JNZ addr	JZ addr	JNC addr	JC addr	JPO addr	JPE addr	JP addr	JM addr
Opcode	C2	CA	D2	DA	E2	EA	F2	FA

7. C. Cond. addr	States—9/18	Flags
—None		

Inst—	CNZ addr	CZ addr	CNC addr	CC addr	CPO addr	CPE addr	CP addr	CM addr
Opcode	C4	CC	D4	DC	E4	EC	F4	FC

8. R. Cond.	States—6/12	Flags
—None		

Inst—	RNZ	RZ	RNC	RC	RPO	RPE	RP	RM
Opcode	C0	C8	D0	D8	E0	E8	F0	F8

I/O and Machine Control Group

1. PUSH rp/PUSH PSW	States—12	Flags
None		

Inst—PUSH	B	D	H	PSW
Opcode	C5	D5	E5	F5

2. POP rp/POP PSW	States—10	Flags
None		

Inst—POP	B	D	H	PSW
Opcode	C1	D1	E1	F1

3. IN/OUT Port No.	States—10	Flags
—None		

Inst—	OUT byte	IN byte
Opcode	D3	DB

4. RIM/SIM	States—4	Flags
—None		
Inst—	RIM	SIM
Opcode	20	30
5. EI/DI	States—4	Flags
—None		
Inst—	DI	EI
Opcode	F3	FB

6. XTHL	Opcode = E3	States—16	Flags
—None			
7. SPHL	Opcode = F9	States—6	Flags
—None			
8. HLT	Opcode = 76	States—5	Flags
—None			
9. NOP	Opcode = 00	States—4	Flags
—None			

Notes: r, r1, r2 — 8-bit register

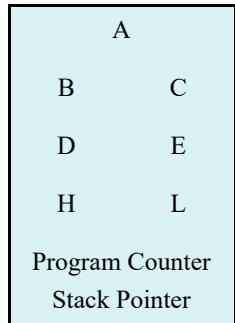
rp — register pair

byte — 8-bit data

dble — 16-bit dat

addr — 16-bit address

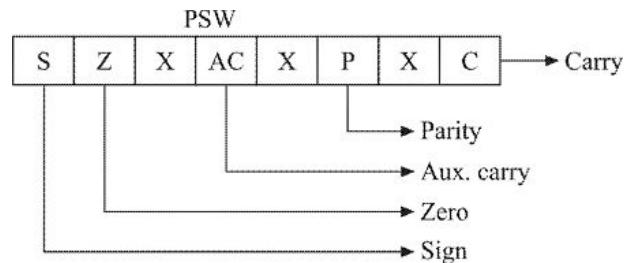
Register Organization



Restart Locations

RST	0	0000
RST	1	0008
RST	2	0010
RST	3	0018
RST	4	0020
RST	5	0028

RST	6	0030
RST	7	0038
TRAP		0024
RST	5.5	002C
RST	6.5	0034
RST	7.5	003C



Appendix II

INTEL 8086 INSTRUCTION SET SUMMARY

Mnemonic and description	Instruction code			
Data Transfer				
MOV = Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH = Push:				
Register/Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment Register	0 0 0 reg 1 1 0			
POP = Pop:				
Register/Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment Register	0 0 0 reg 1 1 1			
XCHG = Exchange:				
Register/Memory with Register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with Accumulator	1 0 0 1 0 reg			

Mnemonic and description	Instruction code			
IN = Input from:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Fixed Port	1 1 1 0 0 1 0 w	port		
Variable Port	1 1 1 0 1 1 0 w			
OUT = Output to:				
Fixed Port	1 1 1 0 0 1 1 w	port		
Variable Port	1 1 1 0 1 1 1 w			
XLAT = Translate Byte to AL	1 1 0 1 0 1 1			
LEA = Load EA to Register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS = Load Pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES = Load Pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF = Load AH with Flags	1 0 0 1 1 1 1			
SAHF = Store AH into Flags	1 0 0 1 1 1 0			
PUSHF = Push Flags	1 0 0 1 1 1 0 0			
POPF = Pop Flags	1 0 0 1 1 1 0 1			
ARITHMETIC				
ADD = Add:				
Reg./Memory with Register to Either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s: w = 0 1
Immediate to Accumulator	0 0 0 0 0 1 0 w	data	data if w = 1	
ADC = Add with Carry:				
Reg./Memory with Register to Either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s: w = 0 1
Immediate to Accumulator	0 0 0 1 0 1 0 w	data	data if w = 1	
INC = Increment:				
Register/Memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA = ASCII Adjust for Add	0 0 1 1 0 1 1 1			
BAA = Decimal Adjust for Add	0 0 1 0 0 1 1 1			
SUB = Subtract:				
Reg./Memory and Register to Either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s: w = 0 1
Immediate from Accumulator	0 0 1 0 1 1 0 w	data	data if w = 1	

Mnemonic and description	Instruction code			
SSB = Subtract with Borrow:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Reg./Memory and Register to Either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s w = 0 1
Immediate from Accumulator	0 0 0 1 1 1 w	data	data if w = 1	
DEC = Decrement:				
Register/Memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG = Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP = Compare:				
Register/Memory and Register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with Register/Memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 0 1
Immediate with Accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
AAS = ASCII Adjust for Subtract	0 0 1 1 1 1 1			
DAS = Decimal Adjust for Subtract	0 0 1 0 1 1 1			
MUL = Multiply (Unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL = Integer Multiply (Signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM = ASCII Adjust for Multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV = Divide (Unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV = Integer Divide (Signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD = ASCII Adjust for Divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW = Convert Byte toWord	1 0 0 1 1 0 0 0			
CWD = Convert Word to Double Word	1 0 0 1 1 0 0 1			
LOGIC				
NOT = Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL = Shift Logical/Arithmetic Left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR = Shift Logical Right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR = Shift Arithmetic Right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL = Rotate Left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR = Rotate Right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL = Rotate Through Carry Flag Left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR = Rotate Through Carry Right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		

Mnemonic and description	Instruction code			
AND = And:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Reg./Memory and Register to Either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 0 0 1 0 w	data	data if w = 1	
TEST = And Function to Flags, No Result:				
Register/Memory and Register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate Data and Register/Memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate Data and Accumulator	1 0 1 0 1 0 0 w	data	data if w = 1	
OR = Or:				
Reg./Memory and Register to Either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 d w	mod 0 0 1 r/m	data	data if w = 1
Immediate to Accumulator	0 0 0 0 1 1 0 w	data	data if w = 1	
XOR = Exclusive or:				
Reg./Memory and Register to Either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 1 0 1 0 w	data	data if w = 1	
STRING MANIPULATION				
REP = Repeat	1 1 1 1 0 0 1 z			
MOVS = Move Byte/Word	1 0 1 0 0 1 0 w			
CMPS = Compare Byte/Word	1 0 1 0 0 1 1 w			
SCAS = Scan Byte/Word	1 0 1 0 1 1 1 w			
LODS = Load Byte/Word to AL/AX	1 0 1 0 1 1 0 w			
STOS = Stor Byte/Word from AL/AX	1 0 1 0 1 0 1 w			
CONTROL TRANSFER				
CALL = Call:				
Direct within Segment	1 1 1 0 1 0 0 0	disp-low	disp-high	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m		
Direct Intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high	
		seg-low	seg-high	
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m		

Mnemonic and description	Instruction code		
JMP = Unconditional Jump:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within Segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within Segment-Short	1 1 1 0 1 0 1 1	disp	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct Intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	
RET = Return from CALL			
Within Segment	1 1 0 0 0 0 1 1		
Within Seg Adding Immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment Adding Immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ = Jump on Equal/Zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE = Jump on Less/Not Greater or Equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG = Jump on Less or Equal/ Not Greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE = Jump on Below/Not above or Equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA = Jump on Below or Equal/ Not Above	0 1 1 1 0 1 1 0	disp	
JP/JPE = Jump on Parity/Parity Even	0 1 1 1 1 0 1 0	disp	
JO = Jump on Overflow	0 1 1 1 0 0 0 0	disp	
JS = Jump on Sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ = Jump on Not Equal/Not Zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE = Jump on Not Less/Greater or Equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG = Jump on Not Less or Equal/ Greater	0 1 1 1 1 1 1 1	disp	
JNB/JAE = Jump on Not Below/Above or Equal	0 1 1 1 0 0 1 1	disp	
JNBE/JA = Jump on Not Below or Equal/Above	0 1 1 1 0 1 1 1	disp	
JNP/JPO = Jump on Not Par/Par Odd	0 1 1 1 1 0 1 1	disp	
JNO = Jump on Not Overflow	0 1 1 1 0 0 0 1	disp	
JNS = Jump on Not Sign	0 1 1 1 1 0 0 1	disp	
LOOP = Loop CX Times	1 1 1 0 0 0 1 0	disp	
LOOPZ/LOOPE = Loop While Zero/Equal	1 1 1 0 0 0 0 1	disp	

Mnemonic and description	Instruction code
	7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
LOOPNZ/LOOPNE = Loop While Not Zero/Equal	1 1 1 0 0 0 0 0 disp
JCXZ = Jump on CX Zero	1 1 1 0 0 0 1 1 disp
INT = Interrupt	
Type Specified	1 1 0 0 1 1 0 1 type
Type 3	1 1 0 0 1 1 0 0
INTO = Interrupt on Overflow	1 1 0 0 1 1 1 0
IRET = Interrupt Return	1 1 0 0 1 1 1 1
Processor Control	
CLC = Clear Carry	1 1 1 1 1 0 0 0
CMC = Complement Carry	1 1 1 1 0 1 0 1
STC = Set Carry	1 1 1 1 1 0 0 1
CLD = Clear Direction	1 1 1 1 1 1 0 0
STD = Set Direction	1 1 1 1 1 1 0 1
CLI = Clear Interrupt	1 1 1 1 1 0 1 0
STI = Set Interrupt	1 1 1 1 1 0 1 1
HLT = Halt	1 1 1 1 0 1 0 0
WAIT = Wait	1 0 0 1 1 0 1 1
ESC = Escape (to External Device)	1 1 0 1 1 x x x mod x x x r/m
LOCK = Bus Lock Prefix	1 1 1 1 0 0 0 0

Notes:

AL = 8-bit accumulator

AX = 16-bit accumulator

CX = Count register

DS = Data segment

ES = Extra segment

Above/below refers to unsigned value

Greater = more positive;

Less = less positive (more negative) signed values

if d = 1 then “to” reg; if d = 0 then “from” reg

if w = 1 then word instruction; if w = 0 then byte instruction

if mod = 11 then r/m is treated as a REG field

if mod = 00 then DISP = 0*, disp-low and disp-high are absent

if mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent

if mod = 10 then DISP = disp-high; disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
 if r/m = 001 then EA = (BX) + (DI) + DISP
 if r/m = 010 then EA = (BP) + (SI) + DISP
 if r/m = 011 then EA = (BP) + (DI) + DISP
 if r/m = 100 then EA = (SI) + DISP
 if r/m = 101 then EA = (DI) + DISP
 if r/m = 110 then EA = (BP) + DISP*
 if r/m = 111 then EA = (BX) + DISP

DISP follows 2nd byte of instruction (before data if required)

if s w = 01 then 16 bits of immediate data from the operand

if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then “count” = 1; if v = 1 then “count” in (CL)

x = don’t care

z is used for string primitives for comparison with ZF FLAG

Segment Override Prefix

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-bit (w = 1)	8-bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

FLAGS = X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

*except if mod = 00 and r/m = 110 then EA = disp-high; disp-low.

Appendix III

INTEL 8051 INSTRUCTION SET SUMMARY

Instructions that affect Flag Settings*

Instruction	Flag			Instruction	Flag		
	C	OV	AC		C	OV	AC
ADD	X	X	X	CLR C	0		
ADDC	X	X	X	CPL C	X		
SUBB	X	X	X	ANL C,bit	X		
MUL	0	X		ANL C,/bit	X		
DIV	0	X		ORL C,bit	X		
DA	X			ORL C,/bit	X		
RRC	X			MOV C,bit	X		
RLC	X			CJNE	X		
SETB C	1						

*Note that operations on SFR byte address 208 or bit addresses 209–215 (i.e. the PSW or bits in the PSW) will also affect flag settings.

Notes on Instruction Set and Addressing Modes

Rn	Register R7-R0 of the currently selected Register Bank.
direct	8-bit internal data location's address. This could be an Internal Data RAM location (0–127) or an SFR [i.e. I/O port, control register, status register, etc. (128–255)].
@Ri	8-bit internal data RAM location (0–255) addressed indirectly through register R1 or R0.
#data	8-bit constant included in the instruction.
#data 16	16-bit constant included in the instruction
addr 16	16-bit destination address. Used by LCALL and LJMP. A branch can be anywhere within the 64 KB program memory address space.
addr 11	11-bit destination address. Used by ACALL and AJMP. The branch will be within the same 2 KB page of program memory as the first byte of the following instruction.
rel	Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is –128 to +127 bytes relative to first byte of the following instruction.
bit	Direct addressed bit in internal data RAM or special function register.

Mnemonic	Description			Byte	Oscillator period
Data Transfer					
MOV A,Rn	Move register to Accumulator			1	12
MOV A,direct	Move direct byte to Accumulator			2	12
MOV A,@Ri	Move indirect RAM to Accumulator			1	12
MOV A,#data	Move immediate data to Accumulator			2	12

MOV	Rn,A	Move Accumulator to register	1	12
MOV	Rn,direct	Move direct byte to register	2	24
MOV	Rn,#data	Move immediate data to register	2	12
MOV	direct,A	Move Accumulator to direct byte	2	12
MOV	direct,Rn	Move register to direct byte	2	24
MOV	direct,direct	Move direct byte to direct	3	24
MOV	direct,@Ri	Move indirect RAM to direct byte	2	24
MOV	direct,#data	Move immediate data to direct byte	3	24
MOV	@Ri,A	Move Accumulator to indirect RAM	1	12
MOV	@Ri,direct	Move direct byte to indirect RAM	2	24
MOV	@Ri,#data	Move immediate data to indirect RAM	2	12
MOV	DPTR,#data16	Load Data Pointer with a 16-bit constant	3	24
MOVC	A,@A+DPTR	Move Code byte relative to DPTR to Accumulator	1	24
MOVC	A,@A+PC	Move Code byte relative to PC to Accumulator	1	24
MOVX	A,@Ri	Move external RAM (8-bit addr) to Accumulator	1	24
MOVX	A,@DPTR	Move external RAM (16-bit addr) to Accumulator	1	24
MOVX	@Ri,A	Move Accumulator to external RAM (8-bit addr)	1	24
MOVX	@DPTR,A	Move Accumulator to external RAM (16-bit addr)	1	24
PUSH	direct	Push direct byte onto stack	2	24
POP	direct	Pop direct byte from stack	2	24
XCH	A,Rn	Exchange register with Accumulator	1	12
XCH	A,direct	Exchange direct byte with Accumulator	2	12
XCH	A,@Ri	Exchange indirect RAM with Accumulator	1	12
XCHD	A,@Ri	Exchange low-order digit indirect		
		RAM with Accumulator	1	12

Mnemonic	Description		Byte	Oscillator period
Arithmetic Operations				
ADD	A,Rn	Add register to Accumulator	1	12
ADD	A,direct	Add direct byte to Accumulator	2	12
ADD	A,@Ri	Add indirect RAM to Accumulator	1	12
ADD	A,#data	Add immediate data to Accumulator	2	12
ADDC	A,Rn	Add register to Accumulator with carry	1	12
ADDC	A,direct	Add direct byte to Accumulator with carry	2	12
ADDC	A,@Ri	Add indirect RAM to Accumulator with carry	1	12
ADDC	A,#data	Add immediate data to Accumulator with carry	2	12
SUBB	A,Rn	Subtract Register from Accumulator with borrow	1	12
SUBB	A,direct	Subtract direct byte from Accumulator with borrow	2	12
SUBB	A,@Ri	Subtract indirect RAM from Accumulator with borrow	1	12

SUBB	A,#data	Subtract immediate data from Accumulator with borrow	2	12
INC	A	Increment Accumulator	1	12
INC	Rn	Increment register	1	12
INC	direct	Increment direct byte	2	12
INC	@Ri	Increment indirect RAM	1	12
DEC	A	Decrement Accumulator	1	12
DEC	Rn	Decrement Register	1	12
DEC	direct	Decrement direct byte	2	12
DEC	@Ri	Decrement indirect RAM	1	12
INC	DPTR	Increment Data Pointer	1	24
MUL	AB	Multiply A and B	1	48
DIV	AB	Divide A by B	1	48
DA	A	Decimal Adjust Accumulator	1	12

Mnemonic	Description		Byte	Oscillator period
Logical Operations				
ANL	A,Rn	AND Register to Accumulator	1	12
ANL	A,direct	AND direct byte to Accumulator	2	12
ANL	A,@Ri	AND indirect RAM to Accumulator	1	12
ANL	A,#data	AND immediate data to Accumulator	2	12
ANL	direct,A	AND Accumulator to direct byte	2	12
ANL	direct,#data	AND immediate data to direct byte	3	24
ORL	A,Rn	OR register to Accumulator	1	12
ORL	A,direct	OR direct byte to Accumulator	2	12
ORL	A,@Ri	OR indirect RAM to Accumulator	1	12
ORL	A,#data	OR immediate data to Accumulator	2	12
ORL	direct,A	OR Accumulator to direct byte	2	12
ORL	direct,#data	OR immediate data to direct byte	3	24
XRL	A,Rn	Exclusive-OR register to Accumulator	1	12
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	12
XRL	A,@Ri	Exclusive-OR indirect RAM to Accumulator	1	12
XRL	A,#data	Exclusive-OR immediate data to Accumulator	2	12
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	12
XRL	direct,#data	Exclusive-OR immediate data to direct byte	3	24
CLR	A	Clear Accumulator	1	12
CPL	A	Complement Accumulator	1	12
RL	A	Rotate Accumulator left	1	12
RLC	A	Rotate Accumulator left through the carry	1	12
RR	A	Rotate Accumulator right	1	12

RRC	A	Rotate Accumulator right through the carry	1	12
SWAP	A	Swap nibbles within the Accumulator	1	12

Mnemonic	Description		Byte	Oscillator period
Boolean Variable Manipulation				
CLR	C	Clear carry	1	12
CLR	bit	Clear direct bit	2	12
SETB	C	Set carry	1	12
SETB	bit	Set direct bit	2	12
CPL	C	Complement carry	1	12
CPL	bit	Complement direct bit	2	12
ANL	C,bit	AND direct bit to carry	2	24
ANL	C,/bit	AND complement of direct bit to carry	2	24
ORL	C,bit	OR direct bit to carry	2	24
ORL	C,/bit	OR complement of direct bit to carry	2	24
MOV	C,bit	Move direct bit to carry	2	12
MOV	bit,C	Move carry to direct bit	2	24
JC	rel	Jump if carry is set	2	24
JNC	rel	Jump if carry not set	2	24
JB	rel	Jump if direct bit is set	3	24
JNB	rel	Jump if direct bit is not set	3	24
JBC	bit, rel	Jump if direct bit is set and clear bit	3	24
Program Branching				
ACALL	addr11	Absolute subroutine call	2	24
LCALL	addr16	Long subroutine call	3	24
RET		Return from subroutine	1	24
RETI		Return from interrupt	1	24
AJMP	addr11	Absolute jump	2	24
LJMP	addr16	Long jump	3	24
SJMP	rel	Short jump (relative addr)	2	24
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	24
JZ	rel	Jump if Accumulator is zero	2	24
JNZ	rel	Jump if Accumulator is not zero	2	24
CJNE	A,direct,rel	Compare direct byte to Accumulator and jump if not equal	3	24
CJNE	A,#data,rel	Compare immediate to Accumulator and jump if not equal	3	24
CJNE	Rn,#data,rel	Compare immediate to register and jump if not equal	3	24
CJNE	@Ri,#data,rel	Compare immediate to indirect and jump if not equal	3	24

DJNZ	Rn,rel	Decrement register and jump if not zero	2	24
DJNZ	direct,rel	Decrement direct byte and jump if not zero	3	24
NOP		No operation	1	12

Appendix IV

INTEL 8096 INSTRUCTION SET SUMMARY

Mnemonic	Operands	Operation (Note 1)	Flags					Notes
			Z	N	C	V	VT	
ADD/ADDB	2	D \leftarrow D + A	✓	✓	✓	✓	↑	—
ADD/ADDB	3	D \leftarrow B + A	✓	✓	✓	✓	↑	—
ADDC/ADDCB	2	D \leftarrow D + A + C	↓	✓	✓	✓	↑	—
SUB/SUBB	2	D \leftarrow D - A	✓	✓	✓	✓	↑	—
SUB/SUBB	3	D \leftarrow B - A	✓	✓	✓	✓	↑	—
SUBC/SUBCB	2	D \leftarrow D - A + C - 1	↓	✓	✓	✓	↑	—
CMP/CMPB	2	D - A	✓	✓	✓	✓	↑	—
MUL/MULU	2	D, D + 2 \leftarrow D \times A	—	—	—	—	—	? 2
MUL/MULU	3	D, D + 2 \leftarrow B \times A	—	—	—	—	—	? 2
MULB/MULUB	2	D, D + 1 \leftarrow D \times A	—	—	—	—	—	? 3
MULB/MULUB	3	D, D + 1 \leftarrow B \times A	—	—	—	—	—	? 3
DIVU	2	D \leftarrow (D, D + 2)/A, D + 2 \leftarrow remainder	—	—	—	✓	↑	— 2
DIVUB	2	D \leftarrow (D, D + 1)/A, D + 1 \leftarrow remainder	—	—	—	✓	↑	— 3
DIV	2	D \leftarrow (D, D + 2)/A, D + 2 \leftarrow remainder	—	—	—	?	↑	—
DIVB	2	D \leftarrow (D, D + 1)/A, D + 1 \leftarrow remainder	—	—	—	?	↑	—
AND/ANDB	2	D \leftarrow D AND A	✓	✓	0	0	—	—
AND/ANDB	3	D \leftarrow B AND A	✓	✓	0	0	—	—
OR/ORB	2	D \leftarrow D OR A	✓	✓	0	0	—	—
XOR/XORB	2	D \leftarrow D (EX.OR) A	✓	✓	0	0	—	—
LD/LDB	2	D \leftarrow A	—	—	—	—	—	—
ST/STB	2	A \leftarrow D	—	—	—	—	—	—
LDBSE	2	D \leftarrow A; D + 1 \leftarrow SIGN(A)	—	—	—	—	—	3, 4
LDBZE	2	D \leftarrow A; D + 1 \leftarrow 0	—	—	—	—	—	3, 4
PUSH	1	SP \leftarrow SP - 2; (SP) \leftarrow A	—	—	—	—	—	—
POP	1	A \leftarrow (SP); SP \leftarrow SP + 2	—	—	—	—	—	—
PUSHF	0	SP \leftarrow SP - 2; (SP) \leftarrow PSW; PSW \leftarrow 0000H	0	0	0	0	0	0
POPF	0	PSW \leftarrow (SP); SP \leftarrow SP + 2; I \leftarrow ✓	✓	✓	✓	✓	✓	✓
SJMP	1	PC \leftarrow PC + 11-bit offset	—	—	—	—	—	5
LJMP	1	PC \leftarrow PC + 16-bit offset	—	—	—	—	—	5

Mnemonic	Operands	Operation (Note 1)	Flags						Notes
			Z	N	C	V	VT	ST	
BR (indirect)	1	PC \leftarrow (A)	—	—	—	—	—	—	
SCALL	1	SP \leftarrow SP - 2; (SP) \leftarrow PC; PC \leftarrow PC + 11-bit offset	—	—	—	—	—	—	5
LCALL	1	SP \leftarrow SP - 2; (SP) \leftarrow PC; PC \leftarrow PC + 16-bit offset	—	—	—	—	—	—	5
RET	0	PC \leftarrow (SP); SP \leftarrow SP + 2	—	—	—	—	—	—	
J (conditional)	1	PC \leftarrow PC + 8-bit offset (if taken)	—	—	—	—	—	—	5
JC	1	Jump if C = 1	—	—	—	—	—	—	5
JNC	1	Jump if C = 0	—	—	—	—	—	—	5
JB	1	Jump if Z = 1	—	—	—	—	—	—	5

Notes:

1. If the mnemonic ends in "B", a byte operation is performed, otherwise a word operation is done. Operands D, B and A must conform to the alignment rules for the required operand type. D and B are the locations in the Register File; A can be located anywhere in memory.
2. D, D + 2 are consecutive WORDS in memory; D is DOUBLE-WORD aligned.
3. D, D + 1 are consecutive BYTES in memory; D is WORD aligned.
4. Changes a byte to a word.
5. Offset is a 2's complement number.

Mnemonic	Operands	Operation (Note 1)	Flags						Notes
			Z	N	C	V	VT	ST	
JNE	1	Jump if Z = 0	—	—	—	—	—	—	5
JGE	1	Jump if N = 0	—	—	—	—	—	—	5
JLT	1	Jump if N = 1	—	—	—	—	—	—	5
JGT	1	Jump if N = 0 and Z = 0	—	—	—	—	—	—	5
JLE	1	Jump if N = 1 or Z = 1	—	—	—	—	—	—	5
JH	1	Jump if C = 1 and Z = 0	—	—	—	—	—	—	5
JNH	1	Jump if C = 0 or Z = 1	—	—	—	—	—	—	5
JV	1	Jump if V = 1	—	—	—	—	—	—	5
JNV	1	Jump if V = 0	—	—	—	—	—	—	5
JVT	1	Jump if VT = 1; Clear VT	—	—	—	—	0	—	5
JNVT	1	Jump if VT = 0; Clear VT	—	—	—	—	0	—	5
JST	1	Jump if ST = 1	—	—	—	—	—	—	5
JNST	1	Jump if ST = 0	—	—	—	—	—	—	5
JBS	3	Jump if Specified Bit = 1	—	—	—	—	—	—	5, 6
JBC	3	Jump if Specified Bit = 0	—	—	—	—	—	—	5, 6
DJNZ	1	D \leftarrow D - 1; if D \neq 0 then PC \leftarrow PC + 8-bit offset	—	—	—	—	—	—	5
DEC/DECB	1	D \leftarrow 0 - 1	✓	✓	✓	✓	✓	↑	—
NEG/NEG B	1	D \leftarrow 0 - D	✓	✓	✓	✓	✓	↑	—
INC/INCB	1	D \leftarrow D + 1	✓	✓	✓	✓	✓	↑	—
EXT	1	D \leftarrow D; D + 2 \leftarrow Sign (D)	✓	✓	0	0	—	—	2
EXTB	1	D \leftarrow D; D + 1 \leftarrow Sign (D)	✓	✓	0	0	—	—	3
NOT/NOTB	1	D \leftarrow Logical Not (D)	✓	✓	0	0	—	—	—
CLR/CLRB	1	D \leftarrow 0	1	0	0	0	—	—	—
SHL/SHLB/SHLL	2	C \leftarrow msb ----- lsb \leftarrow 0	✓	?	✓	✓	↑	—	7
SHR/SHRB/SHRL	2	0 \rightarrow msb ----- lsb \rightarrow C	✓	?	✓	0	—	✓	7
SHRA/SHRAB/SHRAL	2	msb \rightarrow msb ----- lsb \rightarrow C	✓	✓	✓	0	—	✓	7
SETC	0	C \leftarrow 1	—	—	1	—	—	—	—
CLRC	0	C \leftarrow 0	—	—	0	—	—	—	—
CLRVT	0	VT \leftarrow 0	—	—	—	—	0	—	—
RST	0	PC \leftarrow 2080H	0	0	0	0	0	0	8
DI	0	Disable All Interrupts (1 \leftarrow 0)	—	—	—	—	—	—	—
EI	0	Enable All Interrupts (1 \leftarrow 1)	—	—	—	—	—	—	—
NOP	0	PC \leftarrow PC + 1	—	—	—	—	—	—	—
SKIP	0	PC \leftarrow PC + 2	—	—	—	—	—	—	—
NORML	2	Left shift till msb = 1; D \leftarrow shift count	✓	?	0	—	—	—	7
TRAP	0	SP \leftarrow SP - 2; (SP) \leftarrow PC PC \leftarrow (2010H)	—	—	—	—	—	—	9

Notes:

1. If the mnemonic ends in "B", a byte operation is performed, otherwise a word operation is done. Operands D, B and A must conform to the alignment rules for the required operand type. D and B are the locations in the Register File; A can be located anywhere in memory.
5. Offset is a 2's complement number.
6. Specified bit is one of the 2048 bits in the register file.
7. The "L" (Long) suffix indicates double-word operation.
8. Initiates a Reset by pulling RESET low. Software should re-initialize all the necessary registers with code starting at 2080H.
9. The assembler will not accept this mnemonic.

Instructions, Opcodes and State Times

Mnemonic	Operands	Direct				Immediate				Indirect ^(*)				Indexed ^(*)				
		Opcode		Bytes	State times	Opcode		Bytes	State times	Opcode		Bytes	State ⁽¹⁾ times	Auto-inc.		Opcode	Bytes	State ⁽¹⁾ times
		Normal	Auto-inc.			Normal	Auto-inc.			Normal	Auto-inc.			Short	Long			
Arithmetic Instructions																		
ADD	2	64	3	4	65	4	5	66	3	6/11	3	7/12	67	4	6/11	5	7/12	
ADD	3	44	4	5	45	5	6	46	4	7/12	4	8/13	47	5	7/12	6	8/13	
ADDB	2	74	3	4	75	3	4	76	3	6/11	3	7/12	77	4	6/11	5	7/12	
ADDB	3	54	4	5	55	4	5	56	4	7/12	4	8/13	57	5	7/12	6	8/13	
ADDC	2	A4	3	4	A5	4	5	A6	3	6/11	3	7/12	A7	4	6/11	5	7/12	
ADDCB	2	B4	3	4	B5	3	4	B6	3	6/11	3	7/12	B7	4	6/11	5	7/12	
SUB	2	68	3	4	69	4	5	6A	3	6/11	3	7/12	6B	4	6/11	5	7/12	
SUB	3	48	4	5	49	5	6	4A	4	7/12	4	8/13	4B	5	7/12	6	8/13	
SUBB	2	78	3	4	79	3	4	7A	3	6/11	3	7/12	7B	4	6/11	5	7/12	
SUBB	3	58	4	5	59	4	5	5A	4	7/12	4	8/13	5B	5	7/12	6	8/13	
SUBC	2	A8	3	4	A9	4	5	AA	3	6/11	3	7/12	AB	4	6/11	5	7/12	
SUBCB	2	B8	3	4	B9	3	4	BA	3	6/11	3	7/12	BB	4	6/11	5	7/12	
CMP	2	88	3	4	89	4	5	8A	3	6/11	3	7/12	8B	4	6/11	5	7/12	
CMPB	2	98	3	4	99	3	4	9A	3	6/11	3	7/12	9B	4	6/11	5	7/12	
MULU	2	6C	3	25	6D	4	26	6E	3	27/32	3	28/33	6F	4	27/32	5	28/33	
MULU	3	4C	4	26	4D	5	27	4E	4	28/33	4	29/34	4F	5	28/33	6	29/34	
MULUB	2	7C	3	17	7D	3	17	7E	3	19/24	3	20/25	7F	4	19/24	5	20/25	
MULUB	3	5C	4	18	5D	4	18	5E	4	20/25	4	21/26	5F	5	20/25	6	21/26	
MUL	2	(2)	4	29	(2)	5	30	(2)	4	31/36	4	32/37	(2)	5	31/36	6	32/37	
MUL	3	(2)	5	30	(2)	6	31	(2)	5	32/37	5	33/38	(2)	6	32/37	7	33/38	
MULB	2	(2)	4	21	(2)	4	21	(2)	4	23/28	4	24/29	(2)	5	23/28	6	24/29	
MULB	3	(2)	5	22	(2)	5	22	(2)	5	24/29	5	25/30	(2)	6	24/29	7	25/30	
DIVU	2	8C	3	25	8D	4	26	8E	3	28/32	3	29/33	8F	4	28/32	5	29/33	
DIVUB	2	9C	3	17	9D	3	17	9E	3	20/24	3	21/25	9F	4	20/24	5	21/25	
DIV	2	(2)	4	29	(2)	5	30	(2)	4	32/36	4	33/37	(2)	5	32/36	6	33/37	
DIVB	2	(2)	4	21	(2)	4	21	(2)	4	24/28	4	25/29	(2)	5	24/28	6	25/29	

Notes:

*Long indexed and Indirect + instructions have identical opcodes with Short indexed and indirect modes, respectively. The second byte of instructions using any indirect or indexed addressing mode specifies the exact mode used. If the second byte is even, use Indirect or Short indexed. If it is odd, use Indirect + or Long indexed. In all cases the second byte of the instruction always specifies an even (word) location for the address referenced.

(1) Number of state times shown for internal/external operands.

(2) The opcodes for signed multiply and divide are the opcodes for the unsigned functions with an "FE" appended as a prefix.

(*) State times shown for 16-bit bus.

Mnemonic	Operands	Direct				Immediate				Indirect Ⓛ				Indexed Ⓜ			
		Opcode	Normal		Auto-inc.		State Ⓛ times	Short		State Ⓛ times	Long		State Ⓛ times	Short		State Ⓛ times	State Ⓛ times Ⓛ
			Bytes	State times	Bytes	State times		Bytes	State times		Bytes	State times		Bytes	State times		
Logical Instructions																	
AND	2	60	3	4	61	4	5	62	3	6/11	3	7/12	63	4	6/11	5	7/12
AND	3	40	4	5	41	5	6	42	4	7/12	4	8/13	43	5	7/12	6	8/13
ANDB	2	70	3	4	71	3	4	72	3	6/11	3	7/12	73	4	6/11	5	7/12
ANDB	3	50	4	5	51	4	5	52	4	7/12	4	8/13	53	5	7/12	6	8/13
OR	2	80	3	4	81	4	5	82	3	6/11	3	7/12	83	4	6/11	5	7/12
ORB	2	90	3	4	91	3	4	92	3	6/11	3	7/12	93	4	6/11	5	7/12
XOR	2	84	3	4	85	4	5	86	3	6/11	3	7/12	87	4	6/11	5	7/12
XORB	2	94	3	4	95	3	4	96	3	6/11	3	7/12	97	4	6/11	5	7/12
Data Transfer Instructions																	
LD	2	A0	3	4	A1	4	5	A2	3	6/11	3	7/12	A3	4	6/11	5	7/12
LDB	2	B0	3	4	B1	3	4	B2	3	6/11	3	7/12	B3	4	6/11	5	7/12
ST	2	C0	3	4	—	—	—	C2	3	7/11	3	8/12	C3	4	7/11	5	8/12
STB	2	C4	3	4	—	—	—	C6	3	7/11	3	8/12	C7	4	7/11	5	8/12
LDBSE	2	BC	3	4	BD	3	4	BE	3	6/11	3	7/12	BF	4	6/11	5	7/12
LDBZE	2	AC	3	4	AD	3	4	AE	3	6/11	3	7/12	AF	4	6/11	5	7/12
Stack Operations (Internal stack)																	
PUSH	1	C8	2	8	C9	3	8	CA	2	11/15	2	12/16	CB	3	11/15	4	12/16
POP	1	CC	2	12	—	—	—	CE	2	14/18	2	14/18	CF	3	14/18	4	14/18
PUSHF	0	F2	1	8	—	—	—	—	—	—	—	—	—	—	—	—	—
POPF	0	F3	1	9	—	—	—	—	—	—	—	—	—	—	—	—	—
Stack Operations (External stack)																	
PUSH	1	C8	2	12	C9	3	12	CA	2	15/19	2	16/20	CB	3	15/19	4	16/20
POP	1	CC	2	14	—	—	—	CE	2	16/20	2	16/20	CF	3	16/20	4	16/20
PUSHF	0	F2	1	12	—	—	—	—	—	—	—	—	—	—	—	—	—
POPF	0	F3	1	13	—	—	—	—	—	—	—	—	—	—	—	—	—
Jumps and Calls																	
Mnemonic	Opcode	Bytes	States	Mnemonic	Opcode	Bytes	States										
LIMP	E7	3	8	LCALL	EF	3	13/16③										
SJMP	20-27④	2	8	SCALL	28-2F④	2	13/16⑤										
BR []	E3	2	8	RET	F0	1	12/16⑤										
				TRAP③	F7	1	21/24										

Notes:

① Number of state times shown for internal/external operands.

③ The assembler does not accept this mnemonic.

④ The least significant 3 bits of the opcode are concatenated with the following 8 bits to form an 11-bit, 2's complement, offset for the relative call or jump.

⑤ State times for stack located internal/external.

⑥ State times shown for 16-bits bus.

Conditional Jumps

All conditional jumps are 2 byte instructions. They require 8 state times if the jump is taken, 4 if it is not. (8)

Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode
JC	DB	JE	DF	JGE	D6	JGT	D2
JNC	D3	JNE	D7	JLT	DE	JLE	DA
JH	D9	JV	DD	JVT	DC	JST	D8
JNH	D1	JNV	D5	JNVT	D4	JNST	D0

Jump on Bit Clear or Bit Set

These instructions are 3-byte instructions. They require 9 state times if the jump is taken, 5 if it is not. (8)

Mnemonic	Bit Number							
	0	1	2	3	4	5	6	7
JBC	30	31	32	33	34	35	36	37
JBS	38	39	3A	3B	3C	3D	3E	3F

Loop Control

Mnemonic	Opcode	Bytes	State Times
DJNZ	E0	3	5/9 STATE TIME (NOT TAKEN/TAKEN)(8)

Single Register Instructions

Mnemonic	Opcode	Bytes	States (8)	Mnemonic	Opcode	Bytes	States (8)
DEC	05	2	4	EXT	06	2	4
DECB	15	2	4	EXTB	16	2	4
NEG	03	2	4	NOT	02	2	4
NEGB	13	2	4	NOTB	12	2	4
INC	07	2	4	CLR	01	2	4
INCB	17	2	4	CLRB	11	2	4

Shift Instructions

Instr mneemonic	Word		Instr mneemonic	Byte		Instr mneemonic	DBL WD		State Times (8)
	OP	B		OP	B		OP	B	
SHL	09	3	SHLB	19	3	SHLL	0D	3	7 + 1 PER SHIFT(7)
SHR	08	3	SHRB	18	3	SHRL	0C	3	7 + 1 PER SHIFT(7)
SHRA	0A	3	SHRAB	1A	3	SHRAL	0E	3	7 + 1 PER SHIFT(7)

Special Control Instructions

Mnemonic	Opcode	Bytes	States (8)	Mnemonic	Opcode	Bytes	States (8)
SETC	F9	1	4	DI	FA	1	4
CLRC	F8	1	4	EI	FB	1	4
CLRV	FC	1	4	NOP	FD	1	4
RST(6)	FF	1	166	SKIP	00	2	4

Normalize

Mnemonic	Opcode	Bytes	State Times
NORML	0F	3	11 + 1 PER SHIFT

Notes:

(6) This instruction takes 2 states to pull RESET low, then holds it low for 2 states to initiate a reset. The reset takes 12 states, at which time the program restarts at location 2080H. If a capacitor is tied to RESET, the pin may take longer to go low and may never reach the V_{OL} specification.

(7) Execution will take at least 8 states, even for 0 shift.

(8) State times shown for 16-bit bus.

INDEX

- 80386, 47, 52, 53, 61
80486, 47, 52, 53
8051, 50, 66, 67, 418, 419, 420, 482, 521, 567, 710
 memory map, 483
 programming of, 450
8051-based control system, 541
8080, 50, 55, 56, 73
8085, 7, 30, 38, 40, 50, 55, 73, 76, 83, 84, 246, 247, 254, 259, 275, 290, 292, 300, 303,
 318, 319, 320, 321, 322, 339, 351, 367, 373, 375, 376, 396, 567, 620, 694
 functional block diagram, 74
 hardware architecture, 73
 interfacing, 271
 machine language program, 30
 Machine State Chart, 9
 signals, 85
8085A, 75
8085A-2, 75
8085AH, 75
8086, 33, 47, 50, 51, 58, 59, 61, 62, 127, 129, 182, 247, 255, 260, 273, 278, 280, 290,
 294, 297, 307, 319, 320, 328, 340, 343, 353, 367, 378, 405, 567, 703
 block diagram, 58
 interfacing to keyboard and display, 272
 programmer's model, 186
 system configurations, 161
8088, 59, 61, 129, 181, 182
 block diagram, 59
8094, 561
8095, 66, 561, 613
8096, 66, 68, 560, 561, 562, 568, 574, 611, 614, 620, 673, 686, 715
 architecture, 560
 I/O interfacing, 587
 interrupts, 590
 memory interfacing, 580
 programming, 595
 special function registers, 568
8097, 66, 561, 613
8205, 246
8212, 79
8253, 245, 313, 314, 315, 318, 319, 320, 321, 373, 375, 376, 378, 390, 393, 394, 398,
 399, 405, 408
 operating modes, 314

read back command, 318
8254, 245, 313, 314, 315, 319, 320
operating modes, 314
8255, 245, 249, 259, 271, 272, 280, 340, 350, 351, 370, 375, 378, 380, 381, 390, 393, 399, 408
PPI, 373
8279, 281, 282, 283, 284, 285, 286, 289, 290, 292, 293, 294, 297, 299, 303, 307, 321
8282, 155, 161, 169, 443, 576
8284, 128, 162
8286, 154, 163, 169
8288, 128, 163
8394, 561, 562, 565
8395, 66, 561, 565, 613, 674
8396, 561, 562, 565
8397, 66, 561, 562, 565, 613, 674

ACALL, 511
AD 7502, 373
Address bus, 35, 36, 58, 76
Address latch enable (ALE), 76, 89, 139, 431, 432, 435, 567, 581
Addressing modes, 23, 40, 99
 direct, 24
 immediate, 24, 100, 189, 488, 491
 memory, 24
 register, 24, 100, 189, 488
 direct, 100
 indirect, 101, 191
Advance I/O device write control, 157
Advance memory write control, 157
AJMP, 512
ALIGN, 33, 198, 199
Analog-to-digital converter (ADC), 214, 215, 345, 347, 348, 349, 351, 387, 388, 390, 393, 395, 565, 595, 622, 670, 676, 685
Application specific integrated circuit, 3
Arithmetic instructions, 106, 499
Arithmetic and logic group, 27
Assembler directives, 31, 32, 197
Assemblers, 34
Assembly language program, 31
ASSUME, 33
Auxiliary carry plug, 135
AX, register, 131, 186

Base plus index register addressing, 192
Base plus index register relative addressing, 194
Base pointer, 134
Base register, 25
Base register plus index register indirect addressing, 488, 492

Bit addressability, 419
Bit-slice processor (AM 2901), 65
BR, 655
Branch group, 655, 695
Branch instructions, 115
Branch prediction, 47, 65
Bus control, 582
Bus controller Intel 8288, 156
Bus cycles, 146
Bus interface logic, 129
Bus interface unit, 58, 127, 128, 129
BUSWIDTH, 573
BX, register, 131, 186

CALL PROC-NAME, 237
Callahan's corollary, 3
Carry flag, 135
CCR, 568, 580, 581, 586
Centronics interface, 361
CJNE, 515, 516
CLK OUT, 75
Clock, 22, 75
 8085, 73, 74
 generator, 37
CMP, 219
CMPB, 644
CMPSB/CMPSW, 220
Code segment register, 132, 172
Compilers, 33
Computer organization, 17, 34
Condition flags, 75, 76, 98, 566, 567, 625
Continuous-path machines, 666
Control bus, 35, 36
Control unit, 20
Control word, 251, 252, 254, 255, 259, 316, 317, 324, 330, 341, 350, 376, 381, 393,
 394
Cross assemblers, 11
CRT terminal interface, 357
CSEG, 632, 633
CX, register, 131, 186
Cycle stealing, 45

Data bus, 35, 36, 76
 time multiplexing of, 77
Data converter, 369, 389
Data registers, 130
Data segment registers, 132
Data transfer, 694

group, 27, 204, 635, 694
instructions, 103, 494
schemes, 40
 asynchronous, 41
 parallel data transfer, 41
 synchronous, 41
serial, 40, 44, 45
Dedicated microprocessor development system, 12
Destination index, 134, 187
Digital-to-analog converter (DAC), 335, 336, 337, 340, 350
 AD 370/371, 336
Direct addressing, 100, 488
Direction flag, 50, 135
Dirty bit, 53
DIV, 437, 503
DIVU, 643
DJNZ, 516, 657
DMA controller, 59
Documentation, 11
DPTR, 428, 438, 493, 498, 501
DRAM controller, 48
DSEG, 632
DUP, 198
DX, register, 131, 186
Effective address, 188
End-of-conversion signal, 346, 350, 355, 390, 393, 399
ENDM, 201
ENDP, 201
ENDS, 200
EVEN, 198
Execution unit, 58, 127, 128, 129
External memory addressing, 142, 143
Extra segment register, 132
EXTINT, 593
EXTRN, 202

Flag register, 129, 130, 134, 175

GROUP, 200

Halt state, 180
Hardware design, 6
High memory bank, 577, 578
High-order memory bank, 142, 166
High speed input, 595, 598, 599, 622
High speed output, 595, 600
HLDA, 91, 92, 177
HOLD, 91, 92, 177

iAPX
386, 61, 62
432, 61
486, 62
Idle mode, 420
IE, 475, 476
Immediate, 626
Immediate references, 629
IMUL, 218
IN/INW PORT, 209
In-circuit emulator, 14
Index register, 25, 129, 130, 134
Indirect addressing, 490, 626
Indirect references, 627, 628
I/O mapped I/O interface, 38, 39
I/O ports, generation of, 246
I/O read machine cycle, 90
Instruction
decoder, 20, 129
event sequence timing diagram, 87
execution, 21, 87
fetch, 21, 89
format, 22, 23, 99
pipeline, 47
pointer, 134, 172, 174, 187
register, 20, 130
set, 27, 102
INT, 242
Intel 4004, 1, 54
Interrupt, 41, 66, 80, 169
acknowledge, 93
acknowledgement, 176
cycle, 168
data transfer, 43
disabling, 81
enable flag, 135
enabling, 81
expansion, 329
flags, 566
I/O, 41, 42, 43
mask register, 81, 591, 598, 610, 612, 616, 625
maskable, 176
masking, 81
mode, 349
non-maskable, 80, 137, 163, 175
on overflow (INTO), 174, 177, 242
operation, 42
request, 42, 80, 93

service routine, 42, 43, 80, 93, 172, 590
single step mode, 173, 174, 175
software, 173
on terminal count, 314
vector table, 172
Interrupt service routines (ISR), 80
INTR, 80, 81, 93, 137, 169, 176, 177, 380, 405
IP, 468
IRET, 175, 243

JB, 513
JBC, 513
JC, 514
JCXZ, 237
JMP, 234
JNB, 514
JNC, 514

Key debouncing, 265
2-key lockout, 284
N-key rollover, 284
Keyboard, 262, 263, 264, 265, 587
Keyboard and display controller, 245, 281
Keyboard interface, 273, 274, 275, 278, 291, 294, 446, 588
LABEL, 198, 199
LAHF, 209
LCALL, 511, 558, 656
LDBSE, 636
LDBZE, 636
LDS, 207
LEA, 207
LENGTH, 199
LES, 512
LJMP, 512
LOCK, 180, 242
LODSB/LODSW, 207
Logic state analyzer, 13
Logical group, 647, 695
Logical instructions, 110, 504
Long-indexed, 626
LOOP, 237
Loop counter, 565
LOOPE/LOOPZ, 237
LOOPNZ/LOOPNE, 237
Low memory bank, 577, 578
Low-order memory bank, 142, 166

Machine control instructions, 121

Machine cycles, 87
Machine language, 29
MACRO, 201
Macros, 201
Maximum mode, 159, 163
 configuration, 8086, 171
 system configuration, 169
Memory, 19, 39
 cache, 45, 47, 48, 54
 capacity miss, 49
 directory, 48
 multilevel, 49, 50
 trace, 49
 victim, 49
 direct access, 43, 44, 83, 91, 177
 interfacing, 77, 79, 164, 442, 580
 main, 19
 management, 50
 linear memory organization, 50
 segments, 50
 read machine cycle, 90
 secondary, 19
 segmentation, 132
 segmented system, 51
 code segment, 51, 127, 132
 data segment, 51, 127, 132
 extra segment, 51, 127, 132
 stack segment, 51, 127, 132
 segments, 33
 virtual, 45, 51, 52, 54, 59, 62
 descriptor table, 53, 54
 multi level page tables, 53
Memory address decoding, 167
Memory Management Unit (MMU), 45, 52, 54
Memory mapped I/O interface, 39, 55
Memory read, 159
Memory write, 160
Microcomputer, 34, 65, 66
Microcomputer kit, 12
Microcontroller, 65, 66
 8-bit, 66, 68
 16-bit, 66
Microprocessor, 1, 17, 34
 advancements, 45
 based design, 3
 based system, 2, 245
 development of, 11
 buses, 36

8-bit, 55
evolution of, 54
interface to ADC,
 asynchronous mode, 346, 347
 synchronous mode, 347, 348, 607
pipelined architecture, 61, 62
pipelining, 45, 47, 57, 61, 65
vs. random logic, 4
16-bit, 57
64-bit, 65
system development cycle, 2
32-bit, 61
Microprogramming, 65
Minimum mode, 139, 146, 148, 162, 247, 252, 253
 configuration, 170
 system configuration, 169
Motorola
 6800, 6802 and 6809, 55
 68000, 60
 68020, 61, 62
 68030, 62, 63
 68040, 52
 68HCII, 68
 MC68031, 52
MOVC, 428, 493, 496, 497
MOVSB/MOVSW, 208
MOVX, 428, 433, 443, 496, 497
MUL, 217, 437, 502, 641
Multiplexer, 256, 257, 369, 389, 390
Multiprocessing, 45, 65
Multiprocessor architectures, 69
Multitasking, 45, 50, 62
 systems, 59

NAME, 202
NC machine, 665, 670, 673, 674
NEG, 229
No Operation, 174
NOT, 228

Object code queue, 129
OFFSET, 199
Opcode
 execute, 87
 fetch, 87
 fetch cycle, 87
 fetch machine cycle, 88
Operand address, 23

Operating system, 15, 34, 51, 52, 59
Operation code, 23
OR, 225, 226
OUT/OUTW, 210
Output enable signal, 346
Overflow flag, 135
Overflow trap, 625
Overlapped instruction fetch and execution, 130

PAGE, 202
Party flag, 135
PCLK, 378, 405
PCON, 438, 461
Pentium processor architecture, 64
Pentium4, 49
PID algorithm, 352, 353, 355, 357
Point-to-point machine, 666
POP, 206, 696
POPF, 206
Power down mode, 420
Power down operation, 479
Power down RAM, 563, 564, 621
Printer interface, 360, 389
PROC, 201
Program control group, 28
Program counter, 20, 42, 75, 84, 89, 566
Program memory, 419
Program status word (PSW), 75, 97, 98, 421, 422, 438, 482, 486, 565, 566, 620, 624
Programmed I/O, 41, 43
Programmer's model, 98, 185, 482, 620
Protection bits, 53
PTR, 202
PUBLIC, 201
PUSH, 206, 496, 696
PUSHF, 207
RALU, 564, 565, 566, 567, 568
Random logic, 3
RCL, 229
RCR, 230
Read control signal, 137
Read Interrupt Mask (RIM), 73, 82, 83
Read-modify-write instructions, 442
Receive interrupt, 456
Reduced instructions set computing, 65
Referenced bit, 53
Register file, 563, 564, 565, 616, 621
Register indirect addressing, 101, 191, 626
Register relative addressing, 193

REPE/REPZ, 208
REPNE/REPNZ, 208
RETI, 466, 512
ROL, 230
ROR, 230
RS-232C, 357, 358, 360
RSEG, 632, 633
RST/VPD, 479
RST 5.5, 80, 81, 93, 169, 375, 398, 403
RST 6.5, 80, 81, 93, 169, 323
RST 7.5, 80, 81, 93, 169, 323
RXD, 431, 441, 456

SAHF, 209
SAL/SHL, 231
Sample-and-hold circuit, 256, 369, 614
SAR, 231
SBB, 216, 217
SBUF, 427, 438, 570, 607, 610, 611
SCON, 438, 456, 460, 463, 464
SEGMENT, 33, 200
Segment registers, 50, 58, 129, 130, 138, 185
Segment selector, 53, 54
Serial Input Data (SID), 83
Serial Output Data (SOD), 83
Set Interrupt Mask (SIM), 73, 83
Seven-segment display
 interface, 280
 interface to 8279, 286
 parallel interface, 268
 serial interface, 268, 269, 270, 589
Seven-segment LED, 267, 331, 448, 449, 588, 589
Shift-group, 650
Short-indexed, 626
SHR, 232
Sign flag, 133
Signature analyzer, 13
Simulators, 15
SJMP, 512, 655
Software design, 6
 editing and assembling, 10
 hierarchy model, 9
 memory, I/O and register organization, 10
 state transition diagram, 7
Source index, 134, 187
Special control group, 660
Special function registers, 419, 421, 424, 482, 484, 560, 562, 563, 568, 620, 621, 622
 B register, 425

control and status registers, 427
 data pointer, 427
SSEG, 632
Stack, 20, 41, 42, 75, 176
 I/O and machine control instructions, 121
Stack group, 659
Stack pointer, 75, 134, 176, 425, 563, 621
 register addressing, 632
Start Convert, 346, 347, 348, 350, 393
States, 87
Stepper motor, 670, 671, 672, 673, 677, 678, 682, 692
Sticky bit, 625
STOSB/STOSW, 208
Straight line cutting machines, 667
String manipulation, 28
SUB, 215, 216
Subroutine call and return instructions, 116

TCON, 438, 451, 454, 475, 476, 479, 530, 554
TEST, 227, 228
Timer, 371
 functions, 389
 mode, 450
Timer/counter, 66, 371, 380, 450, 595
Timer 0, 465
Timer 1, 466, 597, 598, 600, 601, 602, 686
Timer 2, 597, 598, 600, 601, 602
Timer interrupts, 596
Timer overflow, 590, 598
Timer overflow interrupt, 603
Timers, 595
TITLE, 202
TL0, 451, 452
TL1, 451, 452
TMOD, 438, 454, 471, 476, 530, 554
Translation lookaside buffers (TLBs), 49, 53
Transmit interrupt (T1), 456
Transputer, 69
 concurrent processing, 69
 concurrent programming, 69
TRAP, 80, 81, 93, 169
Trap flag, 135, 174
Tristable
 bus, 36
 devices, 37
 lines, 76
 logic gate, 37
TXD, 431, 441, 456

UART, 460
Universal Microprocessor Development System, 15
USART, 358, 359

Vector processing, 65
Virtual address, 52
Virtual address space, 51
Virtual to physical address translation, 49

Wait for test, 181
Wait state, 48, 80, 91
Wait state control, 586
Wait state request, 80, 137
Wait states, 90
Wasted clock cycles, 47
WATCHDOG, 570, 618
Watchdog timer, 66, 618
Water quantity control, 544
Weigh balance system, 247
Write strobe mode, 582
Write strobe with address valid strobe, 585

XCHG, 209
XLAT, 209
XOR, 226, 227

Z80, 50, 55
Z8000, 60
Z8001, 60
Z8002, 60
Zero flag, 135
Zero register addressing, 632