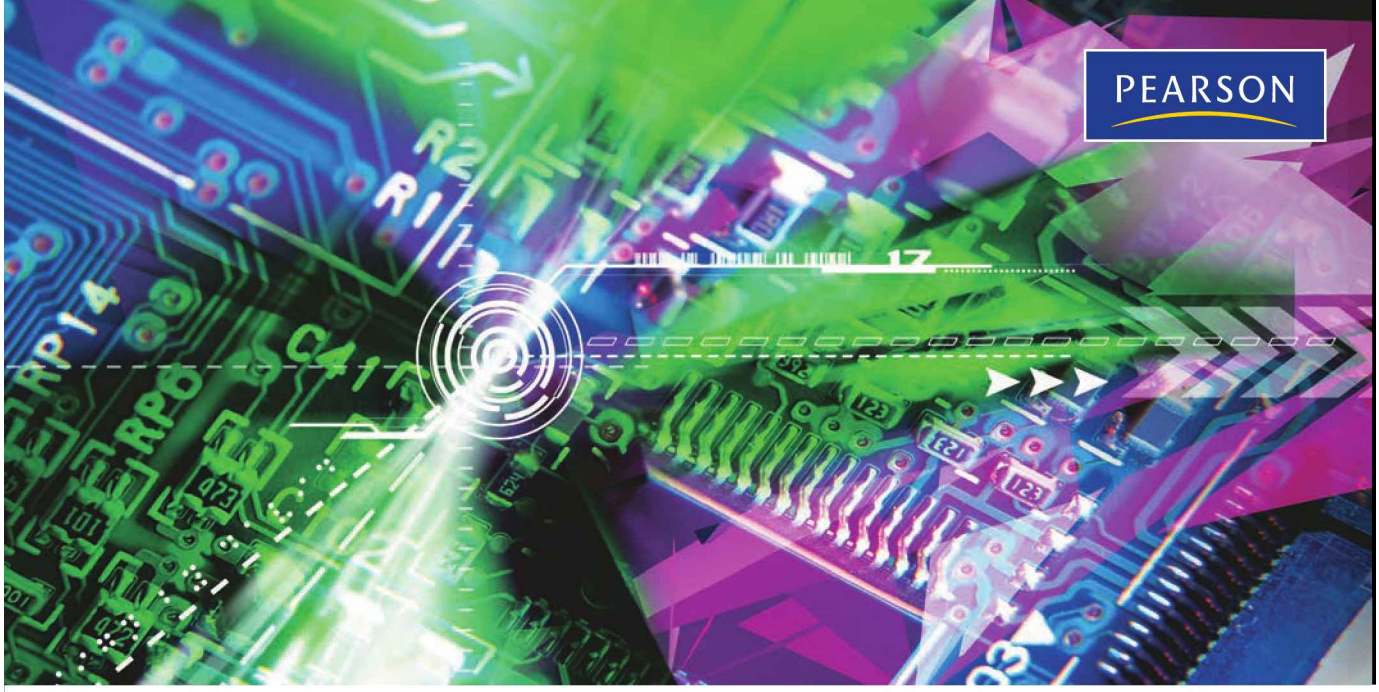
The 8085 Microprocessor

Architecture, Programming and Interfacing

K. Udaya Kumar ▪ B. S. Umashankar



The 8085 Microprocessor

Architecture, Programming and Interfacing

K. Udaya Kumar ▪ B. S. Umashankar

THE 8085 MICROPROCESSOR
Architecture, Programming and Interfacing

K. UDAYA KUMAR,
Principal,
B.N.M. Institute of Technology,
Bangalore, India.

B. S. UMASHANKAR,
Professor,
Department of Computer Science,
B.N.M. Institute of Technology,
Bangalore, India.

PEARSON

ISBN 9788177584554
eISBN 9788131799772

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India
Registered Office: 11 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India

Dedicated to
the Goddess of Learning

This page is intentionally left blank.

Contents

Preface *xi*

Part I

FUNDAMENTALS OF A MICROPROCESSOR 1

1. Evolution of Microprocessors 3
   - 1.1 Early Integrated Circuits 3
   - 1.2 4-Bit Microprocessors 4
   - 1.3 8-Bit Microprocessors 4
   - 1.4 16-Bit Microprocessors 4
   - 1.5 32-Bit Microprocessors 5
   - 1.6 Recent Microprocessors 5
   - 1.7 Microcontrollers and Digital Signal Processors 5

2. Fundamentals of a Computer 7
   - 2.1 Calculator 7
   - 2.2 Computer 8
   - 2.3 Microcomputer 12
   - 2.4 Computer Languages 13
   - Questions 16

3. Number Representation 17
   - 3.1 Unsigned Binary Integers 17
   - 3.2 Signed Binary Integers 18
   - 3.3 Representation of Fractions 23
   - 3.4 Signed Floating Point Numbers 25
   - Questions 25

4. Fundamentals of Microprocessor 27
   - 4.1 History of Microprocessors 27
   - 4.2 Description of 8085 Pins 29
   - 4.3 Programmer's View of 8085: Need for Registers 34
   - 4.4 Accumulator or Register A 35
   - 4.5 Registers B, C, D, E, H, And L 36
   - Questions 36

5. First Assembly Language Program 38
   - 5.1 Problem Statement 38
   - 5.2 About the Microprocessor Kit 41
   - 5.3 Using the Microprocessor Kit in Serial Mode 51
   - Questions 51

6. Data Transfer Group of Instructions 52
   - 6.1 Classification of 8085 Instructions 53
   - 6.2 Instruction Type MVI r, d8 54
   - 6.3 Instruction Type MOV r1, r2 54
   - 6.4 Instruction Type MOV r, M 55
   - 6.5 Instruction Type MOV M, r 56
   - 6.6 Instruction Type LXI rp, d16 56
   - 6.7 Instruction Type MVI m, d8 57
   - 6.8 Instruction Type LDA a16 57
   - 6.9 Instruction Type STA a16 58
   - 6.10 Instruction Type XCHG 58
   - 6.11 Addressing Modes of 8085 59
   - 6.12 Instruction Type LDAX rp 62
   - 6.13 Instruction Type STAX rp 62
   - 6.14 Instruction Type LHLD a16 63
   - 6.15 Instruction Type SHLD a16 63
   - Questions 63

7. Arithmetic Group of Instructions 65
   - 7.1 Instructions to Perform Addition 66
   - 7.2 Instructions to Perform Subtraction 70
   - 7.3 Instruction Type INX rp 73
   - 7.4 Instruction Type DCX rp 74
   - 7.5 Instruction Type DAD rp 74
   - 7.6 Decimal Addition in 8085 75
   - Questions 76

8. Logical Group of Instructions 77
   - 8.1 Instructions to Perform 'AND' Operation 78
   - 8.2 Instructions to Perform 'OR' Operation 79

8.3 Instructions to Perform 'EXCLUSIVE OR' Operation 80
8.4 Instruction to Complement Accumulator 82
8.5 Instructions to Complement/Set 'Cy' Flag 82
8.6 Instructions to Perform Compare Operation 83
8.7 Instructions to Rotate Accumulator 85
Questions 88

9. NOP and Stack Group of Instructions 90

9.1 Stack and The Stack Pointer 90
9.2 Instruction Type POP rp 92
9.3 Instruction Type PUSH rp 93
9.4 Instruction Type LXI SP, d16 94
9.5 Instruction Type SPHL 95
9.6 Instruction Type XTHL 95
9.7 Instruction Type INX SP 95
9.8 Instruction Type DCX SP 96
9.9 Instruction Type DAD SP 96
9.10 Instruction Type NOP 96
Questions 98

10. Branch Group of Instructions 99

10.1 More Details about Program Execution 100
10.2 Unconditional Jump Instructions 102
10.3 Conditional Jump Instructions 104
10.4 Unconditional Call and Return Instructions 107
10.5 Conditional Call Instructions 109
10.6 Conditional Return Instructions 111
10.7 RSTN – Restart Instructions 113
Questions 115

11. Chip Select Logic 117

11.1 Concept of Chip Selection 117
11.2 RAM Chip–Pin Details And Address Range 118
11.3 Multiple Memory Address Range 119
11.4 Working of 74138 Decoder IC 120
11.5 Use of 74138 to Generate Chip Select Logic 121
11.6 Use of 74138 in ALS-SDA-85M Kit 122
Questions 123

12. Addressing of I/O Ports 125

12.1 Need for I/O Ports 125
12.2 IN and OUT Instructions 127
12.3 Memory-Mapped I/O 128
12.4 I/O-Mapped I/O 129
12.5 Comparison of Memory-Mapped I/O and I/O-Mapped I/O 129
Questions 132

13. Architecture of 8085 133

13.1 Details of 8085 Architecture 134
13.2 Instruction Cycle 140
13.3 Comparison of Different Machine Cycles 152
13.4 Memory Speed Requirement 153
13.5 Wait State Generation 160
Questions 161

Part II

ASSEMBLY LANGUAGE PROGRAMS 163

14. Simple Assembly Language Programs 165

14.1 Exchange 10 Bytes 165
14.2 Add two Multi-Byte Numbers 169
14.3 Add two Multi-Byte BCD Numbers 171
14.4 Block Movement without Overlap 174
14.5 Block Movement with Overlap 175
14.6 Add N Numbers of Size 8 Bits 178
14.7 Check the Fourth Bit of a Byte 181
14.8 Subtract two Multi-Byte Numbers 182
14.9 Multiply two numbers of Size 8 Bits 184
14.10 Divide a 16-Bit Number by an 8-Bit Number 187
Questions 189

15. Use of PC in Writing and Executing 8085 Programs 190

15.1 Steps Needed to Run an Assembly Language Program 191
15.2 Creation of .ASM File using a Text Editor 195

15.3 Generation of .OBJ File using a Cross-Assembler 195
15.4 Generation of .HEX File using a Linker 197
15.5 Downloading the Machine Code to the Kit 199
15.6 Running the Downloaded Program on the Kit 201
15.7 Running the Program using the PC as a Terminal 201
Questions 204

16. Additional Assembly Language Programs 205
16.1 Search for a Number using Linear Search 206
16.2 Find the Smallest Number 208
16.3 Compute the HCF of Two 8-Bit Numbers 210
16.4 Check for '2 out of 5' Code 212
16.5 Convert ASCII to Binary 214
16.6 Convert Binary to ASCII 216
16.7 Convert BCD to Binary 218
16.8 Convert Binary to BCD 221
16.9 Check for Palindrome 228
16.10 Compute the LCM of Two 8-Bit Numbers 230
16.11 Sort Numbers using Bubble Sort 233
16.12 Sort Numbers using Selection Sort 235
16.13 Simulate Decimal up Counter 237
16.14 Simulate Decimal down Counter 240
16.15 Display Alternately 00 and FF in the Data Field 241
16.16 Simulate a Real-Time Clock 243
Questions 246

17. More Complex Assembly Language Programs 247
17.1 Subtract Multi-Byte BCD Numbers 248
17.2 Convert 16-Bit Binary to BCD 250
17.3 Do an operation on Two Numbers Based on the Value of X 252
17.4 Do an Operation on Two BCD Numbers Based on the Value of X 255
17.5 Bubble Sort in Ascending/Descending Order as per Choice 259
17.6 Selection Sort in Ascending/Descending Order as per Choice 263
17.7 Add Contents of N Word Locations 266
17.8 Multiply Two 8-Bit Numbers (Shift and Add Method) 268
17.9 Multiply two 2-Digit BCD Numbers 270
17.10 Multiply two 16-Bit Binary Numbers 272
Questions 276

Part III

PROGRAMMABLE AND NON-PROGRAMMABLE I/O PORTS 275

18. Interrupts In 8085 277
18.1 Data Transfer Schemes 278
18.2 General Discussion about 8085 Interrupts 283
18.3 EI and DI Instructions 285
18.4 INTR and INTA* Pins 288
18.5 RST5.5 and RST6.5 Pins 291
18.6 RST7.5 Pin 292
18.7 Trap Interrupt Pin 293
18.8 Execution of 'DAD rp' Instruction 296
18.9 SIM and RIM Instructions 297
18.10 HLT Instruction 302
18.11 Programs using Interrupts 302
Questions 310

19. 8212 Non-Programmable 8-Bit I/O Port 311
19.1 Working of 8212 311
19.2 Applications of 8212 315
Questions 322

20. 8255 Programmable Peripheral Interface Chip 323
20.1 Description of 8255 PPI 323
20.2 Operational Modes of 8255 327
20.3 Control Port of 8255 328
20.4 Mode 1–Strobed I/O 331
20.5 Mode 2–Bi-Directional I/O 340
Questions 342

21. Programs using Interface Modules 344

21.1 Description of Logic Controller Interface 344
21.2 Successive Approximation ADC Interface 353
21.3 Dual Slope ADC Interface 356
21.4 Digital to Analog Converter Interface 359
21.5 Stepper Motor Interface 363
Questions 366

Part IV

SUPPORT CHIPS 367

22. Interfacing of I/O Devices 369

22.1 Interfacing 7-Segment Display 370
22.2 Display Interface using Serial Transfer 374
22.3 Interfacing a Simple Keyboard 377
22.4 Interfacing a Matrix Keyboard 380
22.5 Description of Matrix Keyboard Interface 381
22.6 Intel 8279 Keyboard And Display Controller 384
22.7 Programs using 8279 402
Questions 414

23. Intel 8259A—Programmable Interrupt Controller 416

23.1 Need for an Interrupt Controller 417
23.2 Overview of the Working of 8259 419
23.3 Pins of 8259 421
23.4 Registers used in 8259 422
23.5 Programming the 8259 with no Slaves 424
23.6 Programming the 8259 with Slaves 436
23.7 Use of 8259 in an 8086-Based System 439
23.8 Architecture of 8259 439
Questions 440

24. Intel 8257—Programmable DMA Controller 442

24.1 Concept of Direct Memory Access (DMA) 442
24.2 Need for DMA Data Transfer 443
24.3 Description of 8257 DMA Controller Chip 444
24.4 Programming the 8257 446
24.5 Description of the Pins Of 8257 452
24.6 Working of the 8257 DMA Controller 456
24.7 State Diagram of 8085 457
Questions 460

25. Intel 8253—Programmable Interval Timer 461

25.1 Need for Programmable Interval Timer 461
25.2 Description of 8253 Timer 462
25.3 Programming the 8253 463
25.4 Mode 0–Interrupt On Terminal Count 467
25.5 Mode 1–Re-Triggerable Mono-Stable Multi 468
25.6 Mode 2–Rate Generator 469
25.7 Mode 3–Square Wave Generator 471
25.8 Mode 4–Software-Triggered Strobe 472
28.9 Mode 5–Hardware-Triggered Strobe 473
28.10 Use of 8253 in ALS-SDA-85 Kit 475
Questions 475

26. Intel 8251A—Universal Synchronous Asynchronous Receiver Transmitter (USART) 477

26.1 Need for USART 477
26.2 Asynchronous Transmission 478
26.3 Asynchronous Reception 481
26.4 Synchronous Transmission 483
26.5 Synchronous Reception 484
26.6 Pin Description of 8251 USART 484
26.7 Programming the 8251 488
26.8 Use of SOD Pin of 8085 for Serial Transfer 492
Questions 493

27. Zilog Z-80 Microprocessor 495

27.1 Comparison of Intel 8080 with Intel 8085 496
27.2 Programmer's View of Z-80 497
27.3 Special Features of Z-80 498

27.4 Addressing Modes of Z-80 499
27.5 Special Instruction Types 506
27.6 Pins of Z-80 517
27.7 Interrupt Structure in Z-80 519
27.8 Programming Examples 524
27.9 Instruction Set Summary 527
Questions 528

28. Motorola M6800 Microprocessor 529

28.1 Pin Description of 6800 530
28.2 Programmer's View of 6800 531
28.3 Addressing Modes of 6800 533
28.4 Instruction Set of 6800 536
28.5 Interrupts of 6800 540
28.6 Programming Examples 542
Questions 545

29. 8051 Microcontroller 546

29.1 Main Features of Intel 8051 547
29.2 Functional Blocks of Intel 8051 548
29.3 Program Memory Structure 550
29.4 Data Memory Structure 551
29.5 Programmer's View of 8051 556
29.6 Addressing Modes of 8051 557
29.7 Instruction Set of 8051 560
29.8 Programming Examples 568
Questions 573

30. Advanced Topics in 8051 574

30.1 Interrupt Structure of 8051 575
30.2 Timers of 8051 579
30.3 Serial Interface 584
30.4 Structure and Operation of Ports 591
30.5 Power Saving Modes of 8051 595
30.6 Programming of EPROM in 8751BH 597
Questions 600

Bibliography 601
Index 603

This page is intentionally left blank.

Preface

Microprocessors, microcontrollers, and digital signal processor chips are used in business machines, automotive electronics, home appliances, electronic toys, and a variety of industrial applications. In this book, we confine ourselves to the study of 8-bit microprocessors Intel 8085, Zilog Z-80 and Motorola 6800, as well as the popular 8-bit microcontroller—the Intel 8051.

This book has been written after teaching the subject of microprocessors for more than two decades, keeping in mind the difficulties faced by students in grasping the subject. We have presented the material in a lucid language, using short, simple sentences to facilitate easy reading and understanding. Each concept has been articulated with a number of examples with emphasis on clarity, in a logical sequence. To this end, the book is divided into four parts. The first part consists of Chapters 1 to 13, and deals with the fundamentals of a microprocessor. Chapters 14 to17 make up the second part, and focuses on assembly language programs. The programmable and non-programmable ports are examined in part three from Chapters 18 to 21, while the concluding portion of the book, consisting of Chapters 22 to 30 deals with support chips.

Chapter 1 introduces the developments in electronics starting with the transistor and the early integrated circuits and provides an insight into the evolution of microprocessors, microcontrollers and digital signal processors.

Chapter 2 familiarizes students with the various parts of a computer, their main functions and the evolution of computer languages.

Chapter 3 explains clearly the unsigned and the various signed number representations for integers and provides an overview of signed floating-point numbers.

Chapter 4 touches upon the history of the microprocessor and deals with the fundamentals of the 8085 microprocessor, which is the main focus of this book. The various registers and the programmer's view of 8085 are also introduced here.

Chapter 5 describes a typical 8085-microprocessor kit and its usage by indicating the steps needed to write and execute a simple assembly language program.

Chapter 6 gives the classification of 8085 instructions and elaborates on the data transfer group of instructions with meaningful examples. The various addressing modes of 8085 are also explained.

Chapter 7 deals with the arithmetic group of instructions and explains the various flags used in the 8085 microprocessor.

Chapters 8 to 10 focus on the logical, stack, and branch group of instructions respectively, explaining them with suitable examples.

Chapter 11 dwells on the concept of chip selection and the use of 74138 to generate chip select logic.

Chapter 12 discusses the need for I/O ports, their addressing and compares I/O mapped I/O with memory mapped I/O.

Chapter 13 furnishes a detailed architecture of 8085, and explains the various machine cycles needed for executing a variety of instructions.

Chapter 14 explains simple assembly language programs that are executed on a microprocessor kit and also illustrates some of the commonly used monitor routines.

Chapter 15 brings out the use of a personal computer in writing an assembly language program, translating it to machine language using an assembler, and then downloading it to the microprocessor kit for execution.

Chapter 16 deals with complex assembly language programs. For these programs students have to use the PC to enter the program, do the translation using the assembler, download the machine code to the microprocessor kit, and run the program using the commands issued by the PC in serial mode.

Chapter 17 is about more complex assembly language problems. For each of these problems, the flowchart and the program are provided along with trace for test data. This simplifies the understanding of the given solution.

Chapter 18 expounds on data transfer schemes and discusses in detail about the use of interrupts in the 8085 microprocessor. The interrupt related instructions are explained here, and we look at a number of assembly language programs that make use of interrupts.

Chapter 19 presents a detailed explanation of the working and application of the Intel 8212—a non-programmable I/O port.

Chapter 20 is about the popular Intel 8255—a programmable peripheral interface chip. The description, operational modes and the control words are delineated.

Chapter 21 describes some of the commonly used interface modules like logic controller, analog-to-digital converter, digital-to-analog converter and stepper motor. A number of interesting programs using these interface modules are illustrated.

Chapter 22 first deals with interfacing 7-segment display and matrix keyboard using latches and tri-state gates. Then the Intel 8279—the programmable keyboard and display controller chip is described at length. A number of useful routines using the 8279 chip are also explained.

Chapter 23 is about the Intel 8259—the programmable interrupt controller. It gives an overview of the working of 8259, and explains the function of its pins and the programming of 8259 with and without slave 8259s.

Chapter 24 covers the programmable DMA Controller—the Intel 8257. In this chapter the concept of Direct Memory Access (DMA), the DMA controller chip and its programming are examined in depth.

Chapter 25 describes the Intel 8253—a programmable interval timer. It explains the need for a programmable timer and succinctly spells out the various modes of operation of 8253.

Chapter 26 examines the Intel 8251—the Universal Synchronous Asynchronous Receiver Transmitter (USART). It explains the asynchronous and synchronous modes of transmission and reception, and describes the programming of the 8251.

Chapter 27 reviews the Zilog Z-80 microprocessor. With an in-depth knowledge of the 8085 microprocessor acquired from the first 26 chapters, students would be in a position to understand the programmer's view, new addressing modes, and the new instruction types available in Z-80. The chapter ends with a few programming examples that provide a critical comparison of the Z-80 and the Intel 8085.

Chapter 28 talks about the M6800 microprocessor from Motorola, which has a very simple architecture compared to 8085 or the Z-80. It describes the pins, the programmer's view, addressing modes, and the instruction set of M6800 and ends with a few programming examples that demonstrate its power in spite of its simplicity.

Chapter 29 is devoted to the popular Intel 8051 microcontroller. It discusses the basics of the 8051 providing details about its functional blocks, the programmer's view, addressing modes, and the instruction set. A number of assembly language programming examples are provided to make students comfortable with the instruction set of 8051.

Chapter 30, the concluding chapter, reviews the advanced topics in 8051. It deals with the interrupt structure, timers, serial interface, structure and operation of ports, and power saving modes of 8051.

The chapter ends with the programming of EPROM in 8751, which is the EPROM version of the 8051 microcontroller.

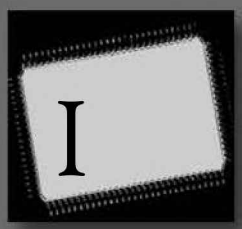
Comments and feedback on the various topics discussed in this book are welcome.

ACKNOWLEDGEMENTS

The authors are grateful for the timely help, encouragement and support extended by Narayan Rao R. Maanay, Secretary, B.N.M. Institute of Technology, as well as Prof. T.J. Rama Murthy, Director, and Dr. K. Ranga, Dean of the institution. They are thankful to the reviewers for their constructive suggestions, which helped in enhancing the contents of this book. Finally, the authors are indebted to their family members for their encouragement and forbearance.

K. Udaya Kumar
B. S. Umashankar

This page is intentionally left blank.



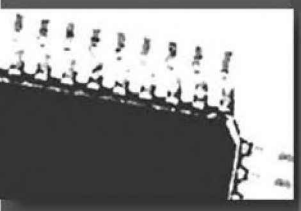
Fundamentals of a Microprocessor

Chapter Heads

1 Evolution of Microprocessors
2 Fundamentals of a Computer
3 Number Representation
4 Fundamentals of Microprocessor
5 First Assembly Language Program
6 Data Transfer Group of Instructions
7 Arithmetic Group of Instructions
8 Logical Group of Instructions
9 NOP and Stack Group of Instructions
10 Branch Group of Instructions
11 Chip Select Logic
12 Addressing of I/O Ports
13 Architecture of 8085

INTRODUCTION

This part comprises of chaps. 1 to 14. Chapters 1 and 2 are introductory chapters which discuss the evolution of microprocessors and the fundamentals of a computer, respectively. The following chapters deal with the framework and internal architecture of a microprocessor.



1 Evolution of Microprocessors

- Early integrated circuits
- 4-bit microprocessors
- 8-bit microprocessors
- 16-bit microprocessors
- 32-bit microprocessors
- Recent microprocessors
- Microcontrollers and digital signal processors

This chapter gives a crisp outline of the various stages in the evolution of today's microprocessors. Explicit information is given right from the integrated circuits through the 4-, 8-, 16- and 32-bit microprocessors to the present-day microprocessors and microcontrollers.

1.1 EARLY INTEGRATED CIRCUITS

The 1939–45 world war posed stringent envorinmental and operation requirements like standardization, miniaturization, reliability, maintainability and the like on electronic communication equipment components. Key improvements took place in the design and manufacture of electronic components. After the war, the semiconductor transistor came into widespread use.

The concept of integrated circuit (IC), also known as 'chip', which integrates a circuit of several electronic components into a solid block was envisaged in 1952. In 1959, the invention of planar process with aluminium metallization by Robert Noyce and Jean Hoerni at Fairchild Semiconductor enabled large-scale production of ICs.

The progress of integration of circuitry was very rapid. The small-scale integration (SSI) chip having digital logic gates circuitry was introduced in 1964. Gordon Earle Moore at Fairchild Semiconductor predicted that the number of transistors on a silicon chip would increase from 50 in 1965 to 65,000 in 1975. It was recognized as his first articulation of Moore's law suggesting that the number of transistors on a chip will double every year. Moore's prediction indeed was true and medium-scale integration (MSI) chip with a complete register circuit appeared in 1968. The large-scale integration (LSI) memory chip (256-bit RAM) was produced by Fairchild in 1970. In 1971, the LSI chips with 1024-bit dynamic random access memory (RAM) and Universal Asynchronous Receiver Transmitter (UART) were developed.

■ 1.2 4-BIT MICROPROCESSORS

The advent of microprocessors was accidental. Intel Corporation founded by Moore and Noyce in 1968 was initially focused on creating semiconductor memory (DRAMs and EPROMs) for digital computers. In 1969, a Japanese calculator manufacturer – Busicom approached Intel with a design for a small calculator, which required 12 custom chips. Ted Hoff, an Intel engineer felt that a general-purpose logic device could replace the separate multiple components. This idea led to the development of the first microprocessor. Microprocessors made a modest beginning as the drivers for calculators.

Federico Faggin and Stanley Mazor realized Ted Hoff's ideas into hardware at Intel. The result was the Intel 4000 family comprising the 4001 (2K ROM), the 4002 (320-bit RAM), the 4003 (10-bit I/O shift-register) and the 4004, a 4-bit central processing unit (CPU). Intel introduced the 4004 microprocessor to the worldwide market on November 15, 1971. It was a 4-bit PMOS chip with 2,300 transistors. It was not truly a general-purpose microprocessor as it was basically designed for a calculator. About the same time, Texas Instruments also developed the 4-bit microprocessor TMS 1000. Texas Instruments is recognized as the inventor and owner of the microprocessor patent.

■ 1.3 8-BIT MICROPROCESSORS

Federico Faggin and his team at Intel designed a chip for controlling a CRT display produced by Computer Terminals Corporation, later called Datapoint. This chip did not meet Datapoint's functional requirement of speed and they decided not to use it. Intel introduced this chip as world's first 8-bit general-purpose microprocessor 8008 in 1972. The Intel 8008 was used in the famous Mark-8 computer kit. On realizing the potential of this product, Intel introduced the improved 8008, the 8080 microprocessor in 1974. The Intel 8080 really created the microprocessor market. The other notable 8-bit microprocessors include Motorola 6800, designed for use in automotive and industrial applications, and Rockwell PPS-8, Signetics 2650 having innovative and powerful instruction set architecture.

With improvements in integration technology, Intel was able to integrate the additional chips required by the 8080, that is, the 8224 clock generator and the 8228 system controller along with the 8080 microprocessor within a single chip – the Intel 8085. The other improved 8-bit microprocessors include Motorola MC6809 designed for high performance, Zilog Z-80 and RCA COSMAC designed for aerospace applications.

In 1975, Moore recalled that his prediction of exponential growth in the complexity of integrated circuits was true. He also forecast a change for the next decade indicating that the pace of complexity increase would slow to a doubling every two years during the maturation of design capabilities.

With increase in processing power the microprocessors dominated as the CPU of digital computers. Earlier to the arrival of microprocessors, CPU was realized from individual SSI chips. The digital computer that uses a single chip microprocessor as its CPU is referred to as a microcomputer.

■ 1.4 16-BIT MICROPROCESSORS

Intel introduced the 16-bit microprocessor 8086 (16-bit bus) in 1978 and 8088 (8-bit bus) in 1979. It had 29,000 transistors. IBM selected the Intel 8088 for their personal computer (IBM-PC) introduced in 1981. Intel released the 16-bit microprocessor 80286 (having 1,34,000 transistors) which was used

as CPU for the advanced technology personal computers (PC-AT) in 1982. It was called Intel 286 and was the first Intel processor that could run all the software written for its predecessor Intel 8088. This backward software compatibility was important for its commercial success. It is important to note that this software compatibility remains a hallmark of Intel's family of microprocessors.

■ 1.5 32-BIT MICROPROCESSORS

In 1985, Intel announced the 80386 a 32-bit microprocessor with 2,75,000 transistors. It supported multitasking. Introduced in 1989, Intel 486 microprocessor was the first to offer a built-in math co-processor. It had 1.2 million transistors.

In 1993, Intel Pentium microprocessor with 3.1 million transistors was introduced. It allowed computers to process real-world data like speech, sound, handwriting and photographic images. The 7.5-million transistor Intel Pentium II microprocessor, introduced in 1997, was designed specifically to process audio, video and graphics data efficiently. Intel Celeron processors range designed for the value PC market segment were released from 1999.

Intel Pentium III processors with 9.5 million transistors designed for advanced imaging, 3D, streaming audio, video and speech recognition applications and Intel Pentium III Xeon processors for workstation and server market segments were introduced in 1999. Intel Pentium IV processors with more than 42 million transistors introduced from 2000 are used in the present PCs. Users can create professional quality movies, deliver TV-like video via the internet, communicate with real-time video and voice, render 3D graphics in real time, quickly encode music for MP3 players and simultaneously run several multimedia applications while connected to the Internet.

Intel Xeon processors introduced from 2001 are targeted for high-performance and mid-range, dual-processor workstations, dual and multiprocessor server configurations coming in the range.

■ 1.6 RECENT MICROPROCESSORS

The Itanium processor is the first in a family of 64-bit products from Intel introduced in 2001. It is well suited for the most demanding enterprise and high-performance computing applications like e-Commerce security transactions, large databases, mechanical computer-aided engineering and sophisticated scientific and engineering computing.

Introduced from 2003, the Intel Pentium M processor, the Intel 855 chipset family and the Intel PRO/Wireless 2100 network connection are the three components of Intel Centrino mobile technology. Intel Centrino mobile technology is designed specifically for portable computing, with built-in wireless local area network (LAN) capability and breakthrough mobile performance. It enables extended battery life and thinner, lighter, mobile computers.

■ 1.7 MICROCONTROLLERS AND DIGITAL SIGNAL PROCESSORS

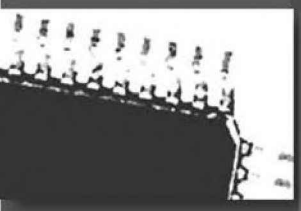
A microcontroller is a highly integrated chip that contains all the components comprising a controller. Typically, this includes a CPU, RAM, some form of read only memory (ROM), input/output (I/O)

ports, timers and so on. Unlike a general-purpose computer, which also includes all of these components, a microcontroller is designed for a very specific task – to control a particular system. As a result, the number of parts can be reduced, which cuts down the cost. Microcontrollers are sometimes called embedded microcontrollers, which just means that they are part of an embedded system – that is, one part of a larger system. One of the popular 8-bit microcontrollers is Intel 8051.

Digital Signal Processor (DSP) is a special-purpose CPU used for digital signal processing applications. Once a signal is converted into digital data, its components can be isolated, analysed and rearranged more easily than in analogue form using various algorithms, such as Fast Fourier Transform. It provides ultrafast instruction sequences, such as shift and add, and multiply and add, which are commonly used in math-intensive signal processing. DSP is used in many fields, including biomedicine, sonar, radar, seismology, audio, cell phones, sound cards, fax machines, modems, hard disks, digital TV, speech and music processing, imaging and communications. It is also used to create the concert hall and surround sound effects in stereo and home theatre equipments. One of the popular DSP chips is TMS320C54X from Texas instruments.

Microprocessors, microcontrollers and DSP chips are used in business machines, automotive electronics, home appliances, electronic toys, variety of industrial applications and the like. Today, there is no field in which microprocessors, microcontrollers or DSP chips have not made an impact.

Semiconductor Industries Association (SIA) has projected annual sales in the year 2006 for microprocessors, microcontrollers and DSPs as $36.4 billion, $12.3 billion and $9 billion, respectively.



2 Fundamentals of a Computer

- Calculator
- Computer
  - *Input devices*
  - *Output devices*
  - *Memory and its classification*
  - *Arithmetic Logic Unit (ALU)*
  - *Control unit and CPU*
  - *Input and output ports*
- Microcomputer
  - *Microprocessor*
  - *Microcontroller*
- Computer languages
  - *Machine language program*
  - *Assembly language program*
  - *High-level language program*
- Questions

This chapter outlines in brief the various components of a computer and their major functions. In the first half of the chapter we deal with the advantages of a computer over a calculator. The chapter ends with a small note on microcomputers and a detailed discussion of the various computer languages along with their merits and demerits.

2.1 CALCULATOR

A calculator is a hand-held equipment using which arithmetic calculations are performed very easily. Block diagram of a calculator is shown in Fig. 2.1.

It will have keypad as an input device. Values needed in a calculation are provided to the calculator using this keypad. There will be an arithmetic unit, which is responsible for performing the needed

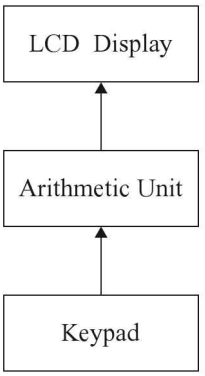


Fig. 2.1
Block diagram of a calculator

calculations. It is a semiconductor chip, which performs the calculations at electronic speeds. Typically it can perform an addition operation in a microsecond.

In an arithmetic operation, say addition, the two inputs are entered using the keypad. The first value that is input goes to the *accumulator*. The accumulator in turn, provides one input to the arithmetic unit. Next the user inputs the operation to be performed by the arithmetic unit (e.g. '+'). Finally, the user inputs the second value to the arithmetic unit using the keypad. Then the arithmetic unit performs the add operation, under the control of the control unit, and displays the result on the LCD panel in about a microsecond after the completion of the operation.

It takes the user about 3 seconds to get this result on the calculator assuming 1 second is needed for inputting each of the values to be added, and 1 second is needed for inputing the operation to be performed. However, the arithmetic unit utilized only about a microsecond for performing the calculation. Thus a calculator becomes slow in the hands as one cannot input values to the calculator at electronic speeds! This problem is overcome in a computer.

■ 2.2 COMPUTER

A computer is generally a desktop equipment, using which arithmetic calculations are performed at a tremendous speed. Nowadays laptop computers as well as hand-held computers have emerged. The blocks that make up a computer are shown in Fig. 2.2.

2.2.1 Input Devices

The input devices are used for supplying program and data to the memory. In other words, the computer system reads the program and the data from the input devices. Most common input devices are the *keyboard* and *mouse*.

2.2.2 Output Devices

Output devices are used for displaying or recording the results computed by the computer. Most common output devices are the *CRT display*, *printer*, and *plotter*.

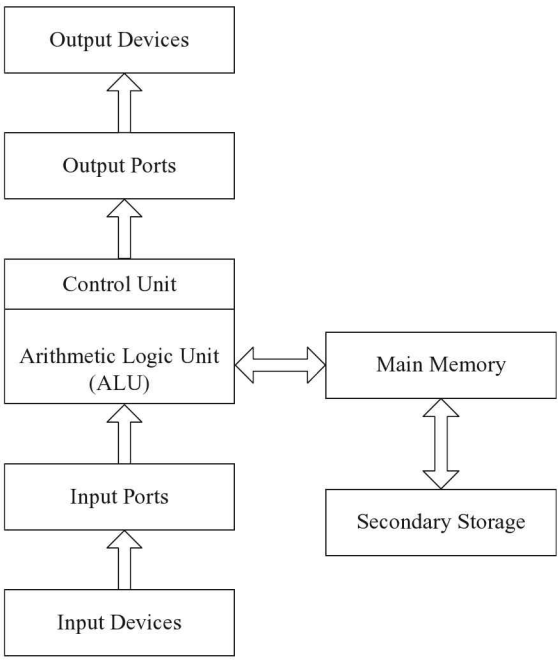


Fig. 2.2
Blocks of a Von Neumann computer

2.2.3 Memory and its Classification

Compared to a calculator, the memory unit is an extra block. In the memory the entire set of operations that are to be performed is written using the keyboard as a sequence of instructions. This sequence of instructions is called *program*. Also, the data on which the program is to operate is entered using the keyboard, and can be stored in memory. Memory can also be used for storing intermediate and final results. This design of a stored program computer where program, data, and result reside in memory is due to Von Neumann.

Memory can be classified into *main memory* and *secondary memory*. Secondary memory is also frequently termed as *auxiliary memory*. Main memory has the drawback of high-cost and low-capacity storage. But its advantage is the high speed of data transfer. The control unit can directly communicate with the main memory but not with the secondary memory. Main memory can be broadly classified into *random access memory* (RAM) and *read only memory* (ROM).

Random access memory (RAM): RAM consists of a number of memory locations where in each location typically 8 bits are stored. (A bit is a binary digit. Thus it can have only one of the two values 0 or 1.) It is possible to read from a RAM location, as well as write to a RAM location. The disadvantage of RAM is that it is volatile. What it means is that, the moment power supply to the RAM is switched off, the information in the RAM will be lost.

Sequential Access: There are two types of accessing information. They are sequential access and random access. In sequential access it is necessary to access information strictly in an order. If there are 100 memory locations, it has to be accessed in the order of 1, 2, 3, ... 100. Thus, it takes least time to access information from location 0 and maximum time to access information from location 100. An example device that employs sequential access is the magnetic tape.

Random Access: In random access technique, it is possible to access a memory location in any order. For example, one can read from the 100 locations in the order of 45, 34, 67, 28, 57, and so on. Second, it takes the same time to read from a memory location irrespective of its position. In a RAM, the method of access is random, and its name in fact is derived based on the method of access.

Read only memory (ROM): ROM also consists of a number of memory locations where in each location typically 8 bits are stored. A ROM also uses random access technique just like a RAM. The advantage of ROM is that it is nonvolatile. What it means is that, even if the power supply to the ROM is switched off, the information in the ROM will not be lost. However, it is only possible to read from a ROM location. Thus, in a computer ROM is used for storing information that should not be lost when power is switched off. But first of all how to store information in a ROM? To answer this question, semiconductor manufacturers provide several versions of ROM whose characteristics are briefly discussed in the following.

Mask-programmed ROM: It derives this name because the information is written to this type of ROM at the time of manufacture using a suitable mask. Once the manufacturer writes this type of ROM, it is not possible to change this information even by the manufacturer. The information entered is permanent. It is cheap compared to the other types of ROMs, when cost per unit quantity is considered. But the user has to place an order for a large number at a time. Thus, ROMs are used in equipments that are produced in large quantities. To give an analogy, mask-programmed ROMs are similar to books that are printed in large volumes.

PROM: It stands for 'programmable read only memory'. The user writes information to this type of ROM using PROM programmer equipment. Once the user writes this type of ROM, it is not possible to change this information anymore. The information entered is permanent. Although it is costlier compared to a mask ROM, it is cheap compared to the other types of ROMs discussed successively in this chapter. The advantage is that the user is free to buy even a single piece of PROM from a local shop. Thus, PROMs are used in equipment that are produced in small quantities. To extend the analogy PROMs are similar to blank notebooks that are written by a user making use of an ink pen.

EPROM: It stands for 'erasable programmable read only memory'. Information is written to this type of ROM by the user using a EPROM programmer equipment. After the user writes this type of ROM, it is possible to change this information a number of times. Thus, although the information entered is not lost when power is switched off, it is possible to erase it and then write new information. Erasing of the EPROM is done by exposure of strong ultraviolet (UV) light from a very close distance on the quartz window of the EPROM chip. Then the contents of the entire EPROM are lost. For this purpose UV light eraser equipment are available. Although it is costlier compared to a mask ROM or a PROM, it is very popular. The user is free to buy even a single piece of EPROM from a local shop. Thus, EPROMs are used during the development phase of the equipment. They are also frequently used in

Type of ROM	*Info. entered by*	*Possible to change info?*	*Cost per chip*	*Min. order quantity*	*Method of erasing info.*	*Application area*
Mask-programmed ROM	Manufacturer	No	Cheapest	Large	Not possible	Large-scale production
PROM	User	No	Costlier	One	Not possible	Medium-scale production
EPROM	User	Yes	Even more costly	One	Using strong UV source	R&D
EEPROM	User	Yes	Costliest	One	Using electrical signals	R&D

equipments that are produced in small quantities. To extend the analogy writing to an EPROM is similar to writing on sand using a stick. If a strong breeze comes everything is erased!

EEPROM or EAPROM: It stands for 'electrically erasable (or alterable) programmable read only memory'. Information is written to this type of ROM by the user using a EPROM programmer equipment. After the user writes this type of ROM, it is possible to change this information a number of times. Thus although the information entered is not lost when power is switched off, it is possible to erase it and then write new information. However, erasing of the EEPROM is done by electrical signals. Then the contents of the EEPROM can be selectively erased. It is costlier compared to the other types of ROMs, and is gaining in popularity. The user is free to buy even a single piece of EEPROM from a local shop. Thus EEPROMs are used during the development phase of the equipment. They are also frequently used in equipments that are produced in small quantities. To extend the analogy EEPROMs are similar to blank notebooks that are written by a user making use of a soft pencil. The information can be selectively rubbed out and new information can be written.

If an EPROM or an EEPROM can be used for reading as well as writing information, and is also nonvolatile, what is the use of RAM? The problem with any type of ROM is that although reading from the ROM is fast, the writing (if permitted) is very slow, and the erasing (if permitted) is extremely slow. Also the writing of the EPROM is not possible when the program is in execution. To give an example, some typical EPROM chips allow reading from a memory location in less than a microsecond, write to a memory location in about 50 ms, and is erased in about 30 minutes.

Secondary memory: Sometimes there is a need to have more amount of information than is possible to store in a main memory. In such cases it is stored in the secondary memory. This type of memory is characterized by virtues of large capacity for storage and low cost per bit of storage. But its disadvantage is its very low speed for access. The control unit does not directly access secondary storage. Typical examples for secondary memory are the *hard disk*, *floppy disk*, and *magnetic tape*.

2.2.4 Arithmetic Logic Unit (ALU)

In a computer, there is an ALU, which is capable of performing logical operations (like AND, OR, Ex-OR, Invert) in addition to the arithmetic operations. The control unit supplies the data required by the ALU from memory, or from input devices, and directs the ALU to perform a specific operation based on the instruction fetched from the memory. ALU is the 'calculator' portion of the computer.

2.2.5 Control Unit and CPU

The control unit fetches one instruction at a time from the main memory, and then executes it. In this execution, it makes use of the ALU, if the instruction execution involves an arithmetic or logical operation (like AND, OR, Ex-OR). This fetching and execution is done at electronic speeds, say in less than a microsecond. Then the control unit fetches and executes the next instruction from the memory, and so on, till the program is completed and the result is output using the output device. In many computers, the control unit and the ALU are integrated into a single block, and this single block is termed as the *central processing unit* (CPU).

The point to be noted here is that the control unit is fetching the instructions from the main memory at electronic speeds, rather than from a human being through a keypad at terribly low speeds. In a calculator we may indicate an operation to be performed by the arithmetic unit in 1 second. In a computer, the control unit fetches and executes more than 1 million instructions at the same time. This is what makes the computer much faster than a calculator. Is a calculator obsolete in view of the above virtues

of the computer? It is not. The reason is simple. If it is required to find the roots of a given quadratic equation one needs only about 1 minute when using a calculator. But using a computer it may take as much as a few hours to develop an error-free program for this problem. It depends on the ability of the programmer. Once the program is developed it takes only a few microseconds to execute the program to obtain the roots of the quadratic equation. Thus a calculator is the choice if the calculation is required to be done only once. A computer is the choice if the calculations are required to be done a number of times with different sets of data.

2.2.6 Input and Output Ports

CPU and the main memory are very fast compared to electromechanical input or output devices like printers, etc. In such a case it is essential that the data lines of the computer is not kept engaged for a long time during communication with input/output (I/O) devices. Otherwise the overall speed of the computer system comes down drastically. So I/O devices are connected to a computer through I/O ports.

For example, to communicate with a printer, the CPU loads the output port connected to the printer at electronic speeds. This is slowly printed by the printer. When the printer has finished printing, the output port requests the CPU for further data. This way, the CPU is allowed to work at its full speed with no degradation in the overall speed of the computer system.

■ 2.3 MICROCOMPUTER

A microcomputer is a small sized, inexpensive, and limited capability computer. It has the same blocks that are present in a computer. Present-day microcomputers are very small in size. They are of the size of a notebook. In the days to come they are bound to become still smaller. They are very cheap so that many individuals can possess them as their personal computers. Because of mass production they are becoming still cheaper. Many early microcomputers were not very powerful. For example, they did not have even a simple multiply instruction in their instruction set. Also, they could work only on unsigned integer data. But present-day microcomputers have not only multiply and divide instructions on unsigned and signed numbers, but are also capable of performing floating point arithmetic operations. In fact they are more powerful than the mini computers and main computers of yesteryear.

2.3.1 Microprocessor

Microprocessor is the CPU part of a microcomputer, and is available as a single integrated circuit. Thus a microprocessor will have the control unit and the ALU of a microcomputer. An example is Intel 8085 microprocessor. In addition to the microprocessor, a microcomputer will have the following:

- ROM/PROM/EPROM/EEPROM for storing program;
- RAM for storing data, intermediate results, and final results;
- I/O devices for communication with the outside world;
- I/O ports for communication with the I/O devices.

Microprocessors are extensively used in the present-day world. Before the advent of the microprocessor, logic design was done by hardware using gates, flip flops, etc. A mini computer was too expensive to think of. With the advent of the microprocessor, logic design using hardware is mostly replaced by design using a microprocessor. This provides 'flexible' instrumentation where just by changing the software it is possible to change the characteristics of the system. Also, new generations of applications have surfaced, which were not thought of earlier because of the prohibitive cost of a mini computer or the complexity of logic design using hardware. A few of the applications using microprocessors are listed below.

- Business applications such as desk-top publishing;
- Industrial applications such as power plant control;
- Measuring instruments such as multi meter;
- Household equipments such as washing machine;
- Medical equipments such as blood pressure monitor;
- Defence equipments such as light combat aircraft;
- Computers such as personal computer.

2.3.2 Microcontroller

It is possible to integrate on a single chip all of the blocks that are needed in a microcomputer, except the I/O devices. Such a chip is termed a microcontroller. An example is Intel 8751. A few of the blocks on the 8751 are:

- 4K $\times$ 8 bits of EPROM;
- 128 $\times$ 8 bits of RAM;
- 4 numbers of 8 bit I/O ports.

It also has timers and facility for serial communication. Microcontrollers are used in a variety of instruments like washing machines, printer sharer, computer keyboards, etc. They are basically used in equipment where the size and cost are required to be very small compared to a microcomputer, and where lots of complex calculations are not needed.

■ 2.4 COMPUTER LANGUAGES

In this section we describe the evolution of computer languages starting from the machine language to the high-level languages. Their merits and demerits are discussed at length.

2.4.1 Machine Language Program

A program can be written using only 0s and 1s. The data can also be specified using only 0s and 1s. Such a program is called machine language program. Machine language was the first in the evolution of computer programming languages. Computer directly understands a program written in the machine language. In fact, even to this day, basically computers understand only the 0s and 1s.

Disadvantages of machine language program: Writing a program in machine language has the following drawbacks.

- It is very tiresome to work with and highly error prone. While writing the program, a 1 and 0 can get interchanged due to typographical error. But then it is very difficult to locate it for correction.
- By a glance through the program, it is very difficult to visualize the function of the program. In fact, it is very difficult to make out whether a particular bit sequence is an instruction in the program, or a data value, or the output result. This is because data, result, and an instruction are represented using 0s and 1s in machine language.
- The same program does not work on another computer by a different manufacturer. This is because machine language is different for different computers. For example, on one computer the bit sequence 10100110 may mean 'add two numbers', and may mean 'subtract two numbers' on another computer. In other words, a program written in machine language is said to be 'not portable'.
- To write a program in machine language one must be highly conversant with the organization and architecture of the computer system being used.

Advantages of machine language program: The only advantages of writing in machine language are:

- Machine language program is executed faster than a program written in a high-level language (high-level language is discussed a little later).
- A translator like compiler is not needed and so results in a cheaper computer system.

To conclude, machine language is rarely used nowadays, except where very high-speed execution is required. It is also used in cheap microcomputer systems.

2.4.2 Assembly Language Program

The next development in the evolution of computer languages was the assembly language. Assembly language uses mnemonics in place of a sequence of 0s and 1s. For example, to add register A and B in a particular computer, assembly language uses the mnemonic 'ADD B' in place of 10000000. Also, assembly language uses symbolic names to denote addresses and data. A number of such examples are dealt with in the successive chapters. Thus writing a program in assembly language has advantages over writing the same in a machine language.

Disadvantages of assembly language program: However, the disadvantages of writing in assembly language are:

- The same program does not work on another computer by a different manufacturer. This is because assembly language is different for different computers. For example, on one computer the mnemonic 'ADD E' may mean 'add contents of A register and E register', and may have no meaning at all on another computer. In other words, a program written in assembly language is also not portable.
- To write a program in assembly language one must be highly conversant with the organization and architecture of the computer system being used.
- An assembler, which is a translator program, is needed for translating the assembly language program into machine code. But each assembly language instruction is translated into only one

instruction in the machine language. As such the assembler program is not a very huge one, and so is quite cheap.

Advantages of assembly language program: The advantages of writing in assembly language are:

- It is less tiresome to work with and much less error prone. While writing the program, if a typographical error occurred due to oversight, it is much easier to locate it for correction. In most cases, the assembler program detects it.
- By a glance through the program it is much easier to visualize the function of the program.
- As far as speed of execution is concerned machine language and assembly language have equal speed of execution. Thus assembly language program is also executed faster than a program written in a high-level language (high-level language is discussed next).

To conclude, assembly language is rarely used nowadays except where very high-speed execution is required. However, it is widely used in microcomputer systems. Both machine language and assembly language are termed as low-level languages.

2.4.3 High-level Language Program

The next development in the evolution of computer languages is the high-level language. Some examples for high-level languages are as follows:

BASIC (for 'Beginners All Purpose Symbolic Instruction Code');
FORTRAN (for 'Formula Translation');
COBOL (for 'COmmon Business Oriented Language');
Pascal (named after the French scientist Blaise Pascal).

High-level languages use English-like language, but with less words and fewer ambiguities. For example, in English the word 'can' can have different meanings based on the context. It is possible to use it as a noun or a verb in a sentence. But high-level languages use only a few words, with each word having a unique meaning.

Disadvantages of high-level languages: However, the disadvantages of writing in high-level languages are:

- A compiler, which is a translator program, is needed for translating the high-level language program into machine code. But each high-level language instruction is translated into more than one instruction in the machine language. As such the compiler program is a huge one and so is quite expensive.
- The code generated by the compiler might not be as compact as written straightaway in low-level language. Thus a program written in high-level language usually takes longer to execute.

Advantages of high-level languages: But writing a program in a high-level language has the following advantages over writing the same in a low-level language.

- It is very easy to learn a high-level language and is a pleasure to work with. While writing the program, if a typographical error occurred due to oversight, it is much easier to locate it for correction. Most of the times the compiler detects it.
- By a glance through the program it is easy to visualize the function of the program.
- The programmer is not required to be familiar with the organization or the architecture of the computer system being used. Thus even school children can work with high-level languages.
- The same program works on any other computer, provided the other computer has a compiler for the language in which the program is written. In other words the programs written in high-level languages are portable.
- Productivity is enormously increased.

To conclude, high-level languages are almost always used nowadays except where very high-speed execution is required.

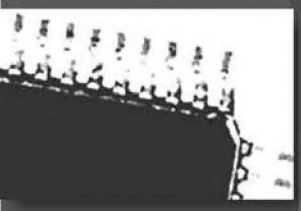
1. What is the difference between a calculator and a computer? What makes a computer faster than a calculator?
2. Briefly explain the blocks that constitute a stored program computer.
3. Name a few input and output devices and their role in a computer.
4. Explain the terms 'random access' and 'sequential access'. Identify the type of access in the following:

   a. RAM,
   b. EPROM,
   c. ROM,
   d. Magnetic tape.

5. Distinguish between main memory and secondary memory. Identify the memory type for the following:

   a. Magnetic disk,
   b. RAM,
   c. Magnetic tape,
   d. EEPROM.

6. Describe the features of various types of ROMs.
7. Describe the role of ALU and control unit in a computer.
8. What is the need for input ports and output ports?
9. Differentiate microprocessor, microcontroller, and microcomputer.
10. Give an overview of the evolution of computer languages, highlighting the merits and demerits of each type of language.



3 Number Representation

- Unsigned binary integers
- Signed binary integers
  - *Sign Magnitude notation*
  - *1's complement notation*
  - *2's complement notation*
- Representation of fractions
  - *2's complement fractions*
- Signed floating point numbers
- Questions

A computer is quite often used for performing computations on numbers. The types of numbers on which the computer is required to perform calculations are: unsigned integers; signed integers; and signed floating point numbers.

In a digital computer everything, whether it is some data, result, or an instruction, has to be represented using only 0s and 1s. This is because the digital computer basically uses transistors that are made to work in the 'Off' state or the 'On' state. The 'Off' state is generally represented as the 0 state and the 'On' state is represented as 1 state. In computers 0s and 1s are called 'bits'. A bit stands for 'BInary digiT'.

3.1 UNSIGNED BINARY INTEGERS

Unsigned binary integers are numbers without any '+' or '−' sign. Examples are: number of books in a library, runs scored by a batsman in a cricket match, etc. Obviously they are unsigned integers like 34,567 and 87. These numbers have to be represented in a computer using only binary notation or bits. Numbers are represented in a computer using a fixed size, like 4, 8, 16, 32 bits, etc. If numbers are represented in a computer using 8 bits, it is said that the computer uses 8-bit word size. Generally, word sizes are a power of 2. In the early days of computing, word sizes were generally 4 bits. These days, it is generally 32 bits or 64 bits. A tabular column of decimal numbers and their equivalent in unsigned binary is shown in the following, assuming a word size of 4 bits.

Number	*Unsigned binary notation*
5	0101
13	1101
0	0000 Minimum number, which is 0
15	1111 Maximum number, which is $(2^4 - 1)$

In this table, just as 13 in decimal notation is $1 \times 10^1 + 3 \times 10^0$, 1101 in binary is $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. From this, it is obvious that if the word size is n bits, the range of $(2^n - 1)$ numbers that can be represented is from 0 to $(2^n - 1)$. A table of word size and the range of unsigned integers that can be represented is shown here.

Word size	*Range for unsigned numbers*
4	0 to 15
8	0 to 255
16	0 to 65535
32	0 to 4,294,967,295

In other words, when the word size is only 4 bits, it is not possible to represent a number like 223. The minimum word size has to be 8 bits to represent the number 223.

■ 3.2 SIGNED BINARY INTEGERS

Signed integers are numbers with a '+' or '−' sign. An example is the list of temperatures (correct to nearest digit) in various cities of the world. Obviously they are signed integers like +34, −15, −23, and +17. These numbers along with their sign have to be represented in a computer using only binary notation or bits.

There are various ways of representing signed numbers in a computer. The simplest way of representing a signed number is the sign magnitude (SM) method.

3.2.1 Sign Magnitude Notation

In this method of representing signed numbers, the most significant bit (MSB) indicates the sign of the number. The number is treated as positive, if the MSB is 0. It is negative if the MSB is 1. The other bits indicate the magnitude of the number.

A tabular column of signed decimal numbers and their equivalent in SM notation follows assuming a word size of 4 bits.

Number	*SM notation*
+5	0 101
−5	1 101
+3	0 011
−3	1 011
+7	0 111 Most positive number, which is $+(2^3 - 1)$
+0	0 000 Notation for +0
−0	1 000 Notation for −0
−7	1 111 Most negative number, which is $-(2^3 - 1)$

From this table, it is obvious that if the word size is n bits, the range of numbers that can be represented is from $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$. A table of word size and the range of SM numbers that can be represented is shown in the following.

Word size	*Range for SM numbers*
4	−7 to +7
8	−127 to +127
16	−32767 to +32767
32	−2147483647 to +2147483647

In other words, when the word size is only 4 bits, it is not possible to represent a number like −123. The minimum word size has to be 8 bits to represent the number −123.

Notice that the bit sequence 1101 corresponds to the unsigned number 13, as well as the number −5 in SM notation. Its value depends only on the way the user or the programmer interprets the bit sequence.

A number is represented inside a computer with the purpose of performing some calculation using that number. The most basic arithmetic operation in a computer is the addition operation. The following examples show the result of adding two signed numbers that are represented inside the computer in SM notation.

Example 1: Add the numbers (+5) and (−3) using a computer. The numbers are assumed to be represented using 4-bit SM notation.

```
1 1 1 1   ← Carries generated during addition
  0101    ← (+5)
 +1011    ← (−3)
 ------
  0000    ← Sum
```

The computer instead of giving the correct answer of +2 = 0010, has given the wrong answer of +0 = 0000! To give the correct answer the computer will have to manipulate the result of addition quite a lot.

Example 2: Add the numbers (−4) and (+2) using a computer. The numbers are assumed to be represented using 4-bit SM notation.

```
0 0 0 0   ← Carries generated during addition
  1100    ← (−4)
 +0010    ← (+2)
 ------
  1110    ← Sum
```

The computer instead of giving the correct answer of −2 = 1010, has given the wrong answer of −6 = 1110! To give the correct answer the computer will have to manipulate the result of addition quite a lot.

Thus to conclude: SM notation is very simple to understand because it is very similar to the conventional way of representing signed numbers. But its disadvantages are:

- There are two notations for 0 (0000 and 1000), which is very inconvenient when the computer wants to test for a 0 result.
- It is not convenient for the computer to perform arithmetic.

Hence, SM notation is generally not used to represent signed numbers inside a computer.

3.2.2 1's Complement Notation

This is another method of representing signed integers in a computer. In this method also, the MSB indicates the sign of the number. The number is treated as positive, if the MSB is 0. It is negative if the MSB is 1. However, the other bits directly indicate the magnitude of the number only when the number is positive. If the number is negative, the other bits signify the 1's complement of the magnitude of the number.

A tabular column of signed decimal numbers and their equivalent in 1's complement notation is shown below, assuming a word size of 4 bits.

Number	*1's complement notation*
+5	0 101
−5	1 010
+3	0 011
−3	1 100
+7	0 111 Most positive number, which is $+(2^3-1)$
+0	0 000 Notation for +0
−0	1 111 Notation for −0
−7	1 000 Most negative number, which is $-(2^3-1)$

From the given table, it is obvious that if the word size is n bits, the range of numbers that can be represented is from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$. A table of word size and the range of 1's complement numbers that can be represented is shown.

Word size	*Range for 1's complement numbers*
4	−7 to +7
8	−127 to +127
16	−32767 to +32767
32	−2147483647 to +2147483647

In other words, when the word size is only 4 bits, it is not possible to represent a number like −123. The minimum word size has to be 8 bits to represent the number −123.

Notice that the bit sequence 1101 corresponds to the following values in different notations.

—1101 is 13 in unsigned notation,
—1101 is −5 in SM notation,
—1101 is −2 in 1's complement notation.

Its value depends only on the way the user or the programmer interprets the bit sequence. Examples for addition of two signed numbers represented using 1's complement notation are shown below.

Example 1: Add the numbers (+5) and (−3) using a computer. The numbers are assumed to be represented using 4-bit 1's complement notation.

```
1 1 0 0   ← Carries generated during addition
  0101    ← (+5)
+ 1100    ← (−3)
-------
  0001    ← Sum
```

The computer instead of giving the correct answer of +2 = 0010, has given the wrong answer of +1 = 0001! However, to get the correct answer the computer will have to simply add to the result the final carry that is generated, as shown in the following.

```
  0001
+    1
------
  0010 = +2
```

Example 2: Add the numbers (−4) and (+2) using a computer. The numbers are assumed to be represented using 4-bit 1's complement notation.

```
0010    ← Carries generated during addition
 1011   ← (−4)
+0010   ← (+2)
------
 1101   ← Sum
```

After the addition of the final carry, the result remains as 1101. This is −2, which is the correct answer. If it is not clear, as to how 1101 is −2, the following explanation should make it clear. In 1 101 the MSB is a 1. It means the number is negative. Then, the remaining bits do not provide the magnitude directly. To solve this problem, just consider 1's complement of 1 101. 1's complement of 1 101 is 0 010, which is +2. Thus, 1 101, which is 1's complement of 0 010 is −2.

Thus to conclude: 1's complement notation is not very simple to understand because it is very much different from the conventional way of representing signed numbers. The other disadvantage is that there are two notations for 0 (0000 and 1111), which is very inconvenient when the computer wants to test for a 0 result.

But, it is quite convenient for the computer to perform arithmetic. To get the correct answer after addition, the result of addition and final carry has to be added up.

Hence, 1's complement notation is also generally not used to represent signed numbers inside a computer.

3.2.3 2's Complement Notation

This is yet another method of representing signed integers in a computer. In this method also, the MSB indicates the sign of the number. The number is treated as positive, if the MSB is 0. It is negative if the MSB is 1. However, the other bits directly indicate the magnitude of the number only when the number is positive. If the number is negative, the other bits signify the 2's complement of the magnitude of the number.

Thus a positive number has the same representation in SM, 1's complement, and 2's complement notations. Only negative numbers are represented differently in these notations.

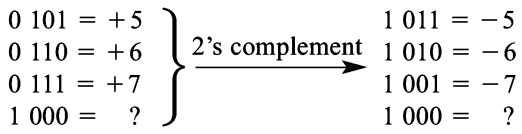
Number	*2's complement notation*
+5	0 101
−5	1 011
+3	0 011
−3	1 101
+7	0 111 Most positive number, which is $+(2^3 - 1)$
0	0 000 Notation for 0
−7	1 001
−8	1 000 Most negative number, which is -2^3

A tabular column of signed decimal numbers and their equivalent in 2's complement notation is shown below, assuming a word size of 4 bits.

Notice that there is a single notation for 0, immaterial of whether it is +0 or −0. One may feel that 0 000 is +0 only, as the MSB in this case is a 0. But then, the notation for −0 should be the 2's complement of 0 000, which is 1111 + 1 = 0 000 ignoring the carry.

If it is not clear, as to how 1000 is −8, the following explanation should make it clear. In 1 000 the MSB is a 1. It means the number is negative. Then, the remaining bits do not provide the magnitude directly. To solve this problem, just consider 2's complement of 1 000. 2's complement of 1 000 is 0 111 + 1 = 7 + 1 = +8. Thus, 1 000 is −8.

An alternative way of arriving at the same conclusion is as follows. Shown in the following discussion is the 2's complement notations for +5, +6, +7 and −5, −6, −7. Then a question mark is placed to indicate the value of 1 000. Using the rules of mathematical induction, 1 000 can be interpreted as either +8 or −8. However, +8 is not correct, as the MSB is 1. Thus, 1 000 has the value −8 in 2's complement notation.



Thus, in 2's complement notation an extra negative number can be represented compared with SM or 1's complement notation. This is because, in 2's complement notation, there is only a single notation for zero, whereas in SM and 1's complement notations there are two notations for 0.

From this, it is obvious that if the word size is n bits, the range of numbers that can be represented is from -2^{n-1} to $+(2^{n-1} - 1)$. A table of word size and the range of 2's complement numbers that can be represented is shown next.

Word size	*Range for 2's complement numbers*
4	−8 to +7
8	−128 to +127
16	−32768 to +32767
32	−2147483648 to +2147483647 = $\pm 2 \times 10^{+9}$ (approx.)

In other words, when the word size is only 4 bits, it is not possible to represent a number like −123. The minimum word size has to be 8 bits to represent the number −123.

Notice that the bit sequence 1101 corresponds to the following values in different notations.

—1101 is 13 in unsigned numbers
—1101 is −5 in SM notation
—1101 is −2 in 1's complement notation
—1101 is −3 in 2's complement notation

Its value depends only on the way the user or the programmer interprets the bit sequence. Examples for addition of two signed numbers represented using 2's complement notation are shown next.

Example 1: Add the numbers (+5) and (−3) using a computer. The numbers are assumed to be represented using 4-bit 2's complement notation.

```
1101    ← Carries generated during addition
 0101   ← (+5)
+1101   ← (−3)
------
 0010   ← Sum
```

Notice that the computer straightaway gives the correct answer of +2 = 0010.

Example 2: Add the numbers (−4) and (+2) using a computer. The numbers are assumed to be represented using 4-bit 2's complement notation.

```
0000    ← Carries generated during addition
 1100   ← (−4)
+0010   ← (+2)
------
 1110   ← Sum
```

This is −2, which is the correct answer. If it is not clear, as to how 1110 is −2, the following explanation should make it clear. In 1 110 the MSB is a 1. It means the number is negative. Then, the remaining bits do not provide the magnitude directly. To solve this problem, just consider 2's complement of 1 110. 2's complement of 1 110 is 0 001 + 1 = 0 010, which is +2. Thus, 1 110, which is 2's complement of 0 010 is −2.

Thus to conclude: 2's complement notation is not very simple to understand because it is very much different from the conventional way of representing signed numbers.

But, its advantage is that there is a single notation for zero, which is very convenient when the computer wants to test for a 0 result. Also, it is very convenient for the computer to perform arithmetic. The addition operation straightaway gives the correct result.

Hence, 2's complement notation is generally used to represent signed numbers inside a computer.

■ 3.3 REPRESENTATION OF FRACTIONS

It may be necessary quite often to represent fractions. For example, it may be needed to represent inside a computer a value like +0.875 or −0.875. This topic is discussed next.

Let us say we have the 4-bit number 1001. Then, what is its value? It has the value 9, −1, −6, or −7 depending on whether it is unsigned, SM number, 1's complement number, or 2's complement number, respectively. Thus the value a given sequence of bits will have depends purely on the interpretation of the user.

So far, only integers were discussed. Thus, a number like 1000 was assumed to have the binary point at the extreme right of the bit sequence. To represent signed fractions, it is necessary to assume the binary point just after the MSB in the bit sequence.

The MS bit specifies the sign of a number. But then, how to specify the binary point inside a computer? This problem is easily solved if all numbers are required to have the binary point at the same position in the bit sequence. For example, if all numbers are required to have the binary point

just after the MS bit, it is just assumed that it exists there! Such numbers where the binary point is assumed to be at a fixed position in the bit sequence are called fixed-point numbers.

Just as there are unsigned, SM, 1's complement, and 2's complement integers, there are also unsigned, SM, 1's complement, and 2's complement fractions. Unsigned fractions will have the assumed binary point at the extreme left. SM, 1's complement, and 2's complement fractions will have this imaginary binary point just to the right of the MS bit.

If the imaginary point is at the extreme right, then the number is an integer. If the imaginary binary point is at the extreme left for an unsigned number, the number is an unsigned fraction. If the binary point is to the immediate right of the MS bit, the number is a signed fraction. If the binary point is in the middle of a bit sequence, the number has an integer and a fractional part.

Only 2's complement fractions, which are the most common, are discussed next.

3.3.1 2's Complement Fractions

For simplicity, the word size is assumed to be 4 bits. It will be generalized to n bits later. The word 0 001 is interpreted as 0.001 if it is a 2's complement fraction. Just as 0.345 decimal has the value $3 \times 10^{-1} + 4 \times 10^{-2} + 5 \times 10^{-3}$, the bit sequence 0.001 will have the value $0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.125$, and is treated as +0.125 as the MS bit is 0.

As another example, what is the value of 1 001, if the interpretation is that it is a 2's complement fraction? It is 1.001 assuming the binary point after the MS bit. As the MS bit is 1, it is a negative number. Then the remaining bits do not specify the magnitude directly. The 2's complement of 1 000 is 0110 + 1 = 0 111. This is a positive fraction with the value $1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0.25 + 0.125 = 0.875$ decimal. Thus, −0.875 is the value of 1 001.

A tabular column of signed decimal fractions and their equivalent in 2's complement notation is shown below, assuming a word size of 4 bits.

Fraction	*2's complement notation*
+0.5	0 100
−0.5	1 100
+0.25	0 001 Smallest magnitude non-zero value is 2^{-3}
−0.25	1 111 Smallest magnitude non-zero value is 2^{-3}
+0.875	0 111 Largest positive magnitude value is $(1-2^{-3})$
−0.875	1 001
−1.0	1 000 Largest negative magnitude value is 1.0
0	0 000

From the above, it is obvious that if the word size is n bits, the smallest magnitude non-zero fraction that can be represented is $2^{-(n-1)}$. A table of word size and the smallest magnitude fraction that can be represented is shown in the following.

Word size	*Smallest magnitude 2's complement fraction*
4	±0.125
8	±0.0078125
16	±0.0000305
32	$\pm 0.5 \times 10^{-9}$

In other words, when the word size is only 4 bits, it is not possible to represent a number like −0.0123 quite accurately. The minimum word size has to be 8 bits to represent the number −0.0123 somewhat accurately. The accuracy is very much improved if the word size is further increased.

■ 3.4 SIGNED FLOATING POINT NUMBERS

In our daily life, we work many times with real numbers, which will have an integer part and a fractional part. An example is 23.456. But this notation is not convenient for representing very large magnitude numbers, like −123456789.4567. This same number can be more conveniently represented in scientific notation as $-1.2345678 \times 10^{+08}$. But this actually stands for −123456780. So there is an error of 9.4567, which forms a very small percentage error.

However, the advantage of this scientific notation is that a very much larger number than $-1.2345678 \times 10^{+08}$ can be represented using same number of digits. For example, $+9.9999999 \times 10^{+99}$ is the largest magnitude positive number using same number of digits.

Similarly, the way we represent real numbers in our daily life is not convenient for representing very small numbers, like +0.00000045678912. This same number can be more conveniently represented in scientific notation as $+4.56789 \times 10^{-07}$. But this actually stands for +0.000000456789. So there is an error of 0.00000000000012, which forms a very small percentage error.

However, the advantage of this scientific notation is that a very much smaller magnitude number than $+4.56789 \times 10^{-07}$ can be represented using same number of digits. For example, -1.00000×10^{-99} is the smallest magnitude negative number using the same number of digits.

Using fixed-point notation, when the word size is as large as 32 bits, the largest magnitude signed integer that can be represented is approximately $\pm 2 \times 10^{+09}$ only. And the smallest non-zero magnitude signed fraction that can be represented is approximately $\pm 0.5 \times 10^{-09}$ only when the word size is 32 bits.

So to represent very large, or very small signed numbers, with a word size of say, 32 bits, a computer uses floating point notation. This may result in small errors, but the error will be negligible, if large number of digits is reserved for the fractional part. But the important thing is that much larger range of numbers can be represented than is possible with fixed-point notation.

A floating-point number like $+1.23 \times 10^4$ can be written in several equivalent ways as $+12.3 \times 10^3$ or $+0.123 \times 10^5$.

Similarly, there are several ways of representing floating point numbers in a computer also. As an example, in the 32-bit floating point notation used in Intel 8087 numeric co-processor, the largest magnitude number is approximately $\pm 5.12 \times 10^{+38}$ and the smallest magnitude number is approximately $\pm 8 \times 10^{-39}$. The interested reader can refer to the book '*Advanced Microprocessors and IBM-PC Assembly Language Programming*' by Udaya kumar and Umashankar (Tata McGraw Hill Publication) for more details.

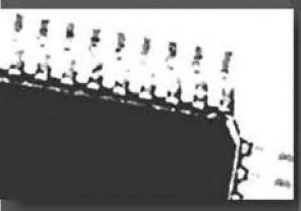
[redacted]

1. Give examples for unsigned integers, signed integers, and floating point numbers.
2. Explain how unsigned numbers are represented in a digital computer. If such numbers are represented using 16 bits, what will be the range?
3. Explain how signed numbers are represented in a digital computer using SM notation. If such numbers are represented using 32 bits, what will be the range?
4. Explain how signed numbers are represented in a digital computer using 1's complement notation. If such numbers are represented using 8 bits, what will be the range?
5. Explain how signed numbers are represented in a digital computer using 2's complement notation. If such numbers are represented using 64 bits, what will be the range?

6. Fill up the following table with the values of the bit sequences in unsigned integer, SM integer, etc.

	Unsigned	*SM*	*1's comp.*	*2's comp.*	*2's comp. fraction*
0110 1100	?	?	?	?	?
1110 0110	?	?	?	?	?
1011 0011	?	?	?	?	?

7. What are the advantages of 2's complement notation over SM and 1's complement notations in representing signed numbers?
8. How can fractions be represented in a computer using fixed-point notation? Explain the 2's complement method of representing fixed-point fractions.



4 Fundamentals of Microprocessor

- History of microprocessors
- Description of 8085 pins
  - *V_{CC}, and V_{SS} pins*
  - *AD_{7-0} pins*
  - *A_{15-8} pins*
  - *ALE pin*
  - *IO/M* pin*
- Programmer's view of 8085: Need for registers
  - *Meaning of programmer's view*
  - Accumulator or register A
  - Registers B, C, D, E, H, and L
  - Questions

4.1 HISTORY OF MICROPROCESSORS

The history of microprocessors dates back to only the year 1971! By this time, the integrated circuits technology was so much developed that it was possible to integrate on a single chip the logic of control unit and ALU. Intel Corporation in the USA announced the first microprocessor in 1971. It was the Intel 4004. It was a 4-bit processor intended for making programmable calculators.

A 4-bit microprocessor receives 4 bits of information from outside the microprocessor, performs the necessary processing on it, and then sends out of the microprocessor a 4-bit result. It will have an ALU that can perform operations on 4-bit numbers. It is said that a 4-bit microprocessor has a word size of 4 bits.

With only 4 bits as the word size, the 4004 could only represent signed numbers in the range -8 to $+7$, which is indeed very small. Hence, it was not really of practical use for arithmetic calculations. However, it found applications in controlling devices. For example, it could be used to switch on or switch off a motor. This was achieved by sending out logic 1 or logic 0 from the microprocessor, and making use of a suitable driver circuitry.

Similarly, it could be used for checking if the temperature of a water bath is above or below a certain desired temperature. This was done by the reading of the logic 1 or 0 value that was developed

by a temperature sensor after proper signal conditioning. The microprocessor can receive this information on a single pin to check the temperature.

Next in the evolution was the Intel 8008, the first 8-bit microprocessor. This was in the year 1972. This was soon followed by Intel 8080, also an 8-bit microprocessor. Intel 8080 was the first commercially popular 8 bit microprocessor. With 8 bits as the word size, it could represent signed numbers in the range of -128 to $+127$. This is also not a good enough range for performing arithmetic calculations. Thus, the 8080 also was used only for control applications.

By this time, some other manufacturers of integrated circuits who were till then manufacturing only components like logic gates, flip flops, shift registers, etc. got fascinated by this new product, and they also released to the market their designs of microprocessor. Notable among them was the 6800 from Motorola and Z-80 from Zilog.

The 6800 design was based on the popular PDP-11 minicomputer of those days. This chip became quite popular for control applications. Z-80 was basically an improvement over the 8080, with a large number of new and powerful instructions. This chip also became very popular in the field of control applications.

In the meanwhile, Intel came out with their improvement over the 8080. This was the Intel 8085. This was only a hardware improvement over 8080. As far as instruction set was concerned, there was basically no improvement. However, the 8085 also became very popular. The 6800, Z-80, and 8085 are all only 8-bit microprocessors, and they rule the 8-bit microprocessor world in control applications.

Around the year 1974, Intel released Intel 8048 family, their *first* microcontroller family. This was later followed by the Intel 8051 series of microcontrollers. Still later Intel released 8096 family of microcontrollers. In the present-day world, a large number of microcontrollers are being used in a variety of consumer products.

Around 1978, Intel released 8086, the first 16-bit microprocessor. With 16-bit word size, it was possible to represent signed numbers in the range of $-32{,}768$ to $+32{,}767$, which is quite a decent range for performing arithmetic calculations. As such, this processor became very popular not only for control applications, but also for number crunching operations. Not to be outdone, Motorola came out with 68000, their 16-bit processor. Zilog released Z-8000, again a 16-bit processor. These are the most popular 16-bit processors.

Then IBM Corporation released the first personal computer (IBM-PC) using 8088 as the CPU developed by Intel. The Intel 8088 has the same instruction set as the 8086. However, it can only receive 8 bits at a time from outside, and send out 8 bits at a time. Internally it is capable of performing 16-bit arithmetic. This chip was designed to make use of the processing power of a 16-bit ALU, while being capable of communicating with the common 8 bit peripheral devices of those days. The IBM-PC became extremely popular, and has now pervaded all walks of our lives. IBM released IBM-PC/XT a little later, using the same Intel 8088. This computer had a hard disk drive in addition to the floppy drives of the earlier PC.

From then on, the development of newer and more powerful processors has been very swift. Intel released 80186, which was mainly used in embedded applications. Then Intel released 80286, which was used in IBM-PC/AT. The processor could be used in multi-user, multi-tasking environment. However, the 80186 and 80286 were still 16-bit processors. In the meanwhile Motorola released 68010, also a 16-bit processor. Around this time Zilog lost the race to capture a major share in the market, with Intel and Motorola emerging as the forerunners to the top spot.

In the early 80s, Intel released the 32-bit processor, the Intel 80386. With 32-bit word size, it was possible to represent signed numbers in the range $\pm 2 \times 10^9$, which is quite a large range for performing arithmetic calculations. If floating point notation is used, it can represent much larger numbers. As such, this processor became very popular as the CPU in computers for number crunching operations. Not to

be outdone, Motorola came out with 68020, their 32-bit processor. Intel released 80486, which was basically a 80386 processor and 80387 numeric co-processor on a single chip. Motorola released 68030. In the early 90s Intel deviated from their naming tradition and released 80586 by the name Pentium processor. It is extremely fast in performing arithmetic calculations and executing instructions. The Pentium 4 released in 2000 has 42 Million transistors worked with a clock frequency of 1.5 GHz and is rated for 1500 MIPS (Million instructions per second)

To conclude, the present-day computers based on microprocessors are already faster than the mini computers and sometimes the main frame computers of yesteryear, and they are available at a small fraction of the cost of such main frame computers. We can hope to see the release of still more powerful and at the same time cheaper microprocessors and microcontrollers in the future.

In this book, Intel 8085 processor will be discussed at great length. Armed with that knowledge, Zilog Z-80, Motorola 6800 processors, and Intel 8051 microcontroller will be discussed quite exhaustively in a few chapters.

■ 4.2 DESCRIPTION OF 8085 PINS

Intel 8085 is fabricated as a 40-pin DIP IC. DIP stands for 'dual inline package'. It means the package will have pins on only two sides, 20 on each side in this case.

Intel manufactures 8085 in several versions, like 8085A, 8085AH, 8085AH-2, and 8085AH-1. The 8085A is fabricated using NMOS technology. It is a variant of MOS (metal oxide semiconductor) technology. It uses *n* channel silicon-gate process. The AH series are more expensive processors, which use high-density MOS (HMOS) for fabrication. They typically consume 20% less power compared to the A series. The recommended internal clock frequency for the various types of 8085 is as follows.

Type	*Recommended clock frequency*
8085A and 8085AH	3 MHz
8085AH-2	5 MHz
8085AH-1	6 MHz

Basically they are same, the only difference being in the speed of operation, or the technology used for fabrication. In this discussion, they are simply referred to as 8085. The pin diagram of 8085 is as shown in fig. 4.1.

In fig. 4.1, the pin number and its associated function is indicated for each of the 40 pins. For example, the diagram indicates that pin number 20 is the V_{ss} pin, which should be connected to ground, and 40 is the V_{cc} pin, which should be connected to +5 V dc supply. A user definitely needs this information, when he is required to wire up a microprocessor in his circuit. However, for the purpose of understanding the working of the processor, only the function of the various pins need to be known. There is no need to know which pin number performs what function. For example, to understand the working of 8085 microprocessor, the user should be aware that it needs a power supply of +5 V dc and ground. It is not necessary to know the pin numbers to which +5 V dc and ground are to be connected. A simplified diagram that does not indicate the pin numbers, but only indicates the function of the various pins, can be called a functional pin diagram. Fig. 4.2 partly provides the functional pin diagram of 8085. It shows only 22 pins. The remaining portion is explained later in the text.

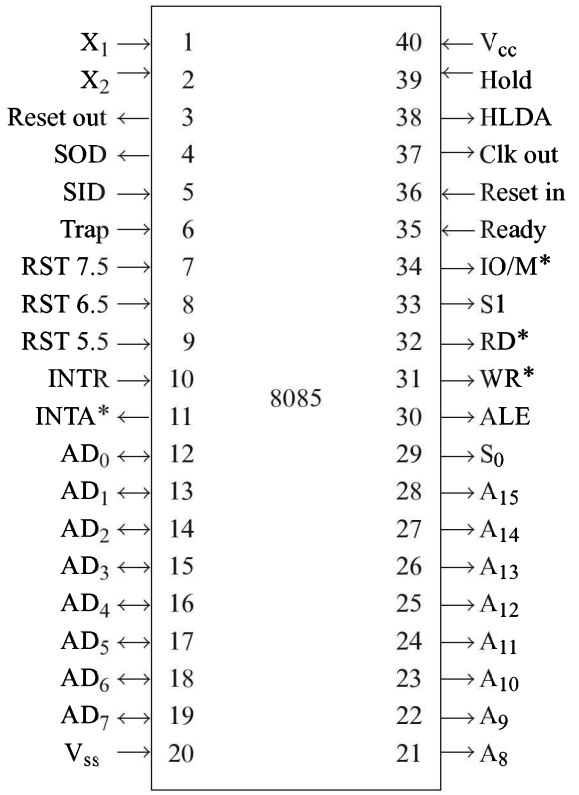


Fig. 4.1
Pin diagram of 8085

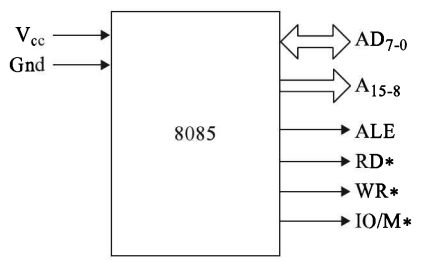


Fig. 4.2
Functional pin diagram
of 8085 (incomplete)

4.2.1 V_{CC} AND V_{SS} PINS

Any IC needs a power supply for its working. Similarly, 8085 needs a power supply of +5 V dc for its working. Pin 40 of 8085 is the V_{cc} pin. It should be connected to +5 V dc power supply. Pin 20 of 8085 is the V_{ss} pin. It should be connected to power supply ground.

4.2.2 AD_{7-0} PINS

Intel 8085 communicates with the outside world with 8 bits at a time. When the 8085 desires to receive 8-bit information, it receives it on the eight pins AD_7, AD_6, ... ,AD_0. These eight pins are

collectively called AD_{7-0}. Their pin numbers are 19, 18, ... , 12, respectively. When the 8085 desires to send out 8-bit information, it sends it out on these same pins.

Thus, these pins are used for receiving as well as sending out information. In other words, they are bi-directional pins. These pins are used for:

- Receiving the program code from memory 8 bits at a time.
- Receiving a data byte (byte = 8 bits) from an input port or from memory.
- Sending out a byte to an output port or to memory

How can the same lines be used for receiving information as well as sending out information? This is not at all a problem, because at any instant of time, the processor is either receiving or sending out information, but not both. They are decided by the signals sent out on RD* and WR* pins, which are discussed next.

RD and WR* Pins : RD**stands for 'Read*'. A '*' in this book after a signal name indicates that the signal is an active low output. Thus RD* is an active low output signal. Another popular convention is to denote RD* as $\overline{RD}$.

RD* is pin number 32 of 8085. When the control unit in the 8085 sends out logic 0 on this pin, it is indicating that it desires to read information. Then the information comes to the processor on the AD_{7-0} pins. In other words, when RD* = 0, AD_{7-0} pins are input pins. When RD* = 1, the 8085 is not interested in reading information.

WR* stands for 'Write*'. WR* is an active low output signal. WR* is pin number 31 of 8085. When the control unit in the 8085 sends out logic 0 on this pin, it is indicating that it desires to write information. Then the information is sent out by the processor on the AD_{7-0} pins. In other words, when WR* = 0, AD_{7-0} pins are output pins. When WR* = 1, the 8085 is not interested in writing or sending out information.

Thus the action performed by the 8085 for various combinations of RD* and WR* are as follows.

*RD**	*WR**	*Action*
0	1	8085 reads information (AD_{7-0} are input pins)
1	0	8085 writes information (AD_{7-0} are output pins)

If the control unit in the 8085 sends out logic 1 on both RD* and WR* simultaneously, it means that the 8085 is not interested in reading or writing at that moment. As an example, when the 8085 is busy with some internal processing it sends out logic 1 on both RD* and WR*.

The control unit will never send out logic 0 on both RD* and WR* simultaneously. If such a thing ever happens, it means the processor has gone crazy, and you better throw it!

4.2.3 A_{15-8} Pins

In memory, the program to be executed is stored. It may occupy a number of locations, depending on the complexity of the program. Each location in memory is used for storing 8 bits of information. There is no point in a memory location having 4 bits or 16 bits, when the 8085 can read or write 8 bits at a time. Also, memory is used for storing data and results, 8 bits in each memory location.

But, it is very inconvenient to indicate memory contents using 8 bits, which is too long. Thus, memory contents are indicated in a shorter form. Suppose, the contents of the first three memory locations

in binary are 0011 0101, 1010 0010, and 0001 1101. The same contents in decimal could be written as 53, 162, and 29. But converting binary number to decimal and vice versa is quite complex. Thus, it is preferred to indicate the same contents in hexadecimal notation as 35H, A2H, and 1DH.

Converting a value in binary to hexadecimal and vice versa is extremely simple. The binary number should just be written in groups of 4 bits, and convert each group of 4 bits into equivalent hexadecimal number. A hexadecimal number is many times called a 'hex' number for short. In this book, the content of a memory location is generally shown in hex for convenience. But, the reader should always remember that a memory location contains 8 bits of information.

When there are a number of memory locations, each location should have a unique address to identify the location. In a microcomputer, even the address has to be specified using only 0s and 1s.

If there are only two memory locations, just 1 bit address is enough to specify the location of interest. If address is 0, location 0 is selected. If address is 1, location 1 is selected.

If there are four memory locations, 2-bit address is needed to specify the location of interest. If address is 00, location 0 is selected. If address is 01, location 1 is selected. If address is 10, location 2 is selected. If address is 11, location 3 is selected.

Similarly, if there are eight memory locations, 3-bit address is needed to specify the location of interest. If address is 000, location 0 is selected, etc. and if address is 111, location 7 is selected.

Thus in general, if there are n address bits, it is possible to specify any memory location out of 2^n memory locations. Intel 8085 is provided with 16 pins for sending out address information. They are $A_{15\text{-}8}$ and $AD_{7\text{-}0}$ pins. On $A_{15\text{-}8}$ the 8085 sends out the MS byte of address. Their pin numbers are 28 to 21, respectively. The A in $A_{15\text{-}8}$ stands for 'Address'. On $AD_{7\text{-}0}$ the LS byte of address is sent out. With 16 address pins the 8085 is capable of addressing any memory location out of $2^{16} = 2^6 \times 2^{10} = 64 \times 1024 = 65{,}536$ locations. In computer jargon, it is termed 64K locations, where $1K = 2^{10} = 1024$.

The addresses of these 65,536 locations in binary number system will be as follows.

```
0000 0000 0000 0000
0000 0000 0000 0001
0000 0000 0000 0010
.
.
.
1111 1111 1111 1111
```

The processor sends out the address in binary as shown previously. But, it is very inconvenient to write down addresses using 16 bits, which is too long. Thus, it is preferable to write down address in a shorter form. The same addresses in decimal could be written as 0, 1, 2, ... , 65,535. But converting a binary address to decimal and vice versa is quite complex. Thus, it is preferred to write down the same addresses in hexadecimal notation as 0000H, 0001H, 0002H, ... , FFFFH. In this book, the addresses are generally shown in hex for convenience. But, the reader should always remember that the processor sends out the address in binary on the 16 pins $A_{15\text{-}8}$ and $AD_{7\text{-}0}$.

Thus memory in a 8085-based microcomputer may be conceptualized as shown in Fig. 4.3.

If the processor is interested in reading information from memory location whose address in hex is 0002H, it sends out 00H on $A_{15\text{-}8}$ and 02H on $AD_{7\text{-}0}$. Then the processor sends out RD* as logic 0 and WR* as logic 1. This results in 1DH, the contents of location 0002H, coming to the processor from memory on $AD_{7\text{-}0}$.

Similarly, if the processor is interested in writing the information 89H to memory location whose address in hex is 4567H, it sends out 45H on $A_{15\text{-}8}$ and 67H on $AD_{7\text{-}0}$. A little later, the processor sends

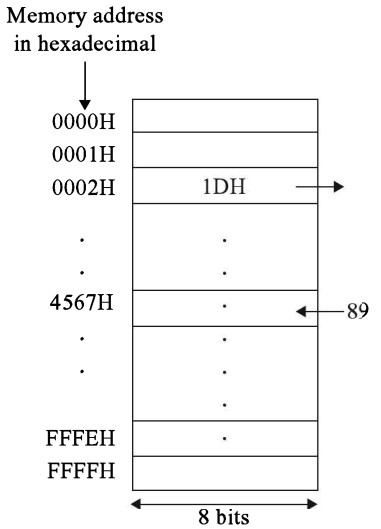


Fig. 4.3
Memory in 8085-based microcomputer

out 89H on AD_{7-0} with RD* as logic 1 and WR* as logic 0. This results in 89H getting written to the location 4567H.

Thus, it is seen that AD_{7-0} is used for sending out LS byte of address, as well as sending or receiving 8 bits of data. In fact, AD in AD_{7-0} stands for 'address data'. This multiplexing of LS byte of address and data has been done in 8085 to save on the number of pins required for the chip.

4.2.4 ALE Pin

If LS byte of address and data are multiplexed on AD_{7-0}, how to know whether information on AD_{7-0} is address or data? This is solved by the ALE signal sent out by the 8085. ALE stands for 'address latch enable'.

At this moment, it is enough to understand that if this signal is at logic 1, 8085 is sending out address information on AD_{7-0}. In other words, when ALE = 1, AD_{7-0} pins act like address pins A_{7-0}. If ALE is at logic 0, the 8085 is sending out or receiving information. In other words, when ALE = 0, AD_{7-0} pins act like data pins D_{7-0}.

Information is received on AD_{7-0} when ALE = 0, if RD* = 0 and WR* = 1. Then AD_{7-0} pins act like the input pins D_{7-0}. Information is sent out on AD_{7-0} when ALE = 0, if WR* = 0 and RD* = 1. Then AD_{7-0} pins act like the output pins D_{7-0}.

4.2.5 IO/M* Pin

It has already been seen that the microprocessor does not directly communicate with an I/O device. It only communicates with I/O ports. As we can have a number of I/O devices in a microcomputer system, we need to have a number of I/O ports in the system. If there are a number of I/O ports in the system, each port must have a unique address, just like each memory location has a unique address. The 8085 sends out the port address on A_{15-8} and AD_{7-0} only, just like sending out a memory address. Then, how to identify whether the address is for memory or for an I/O port? This is solved by the output signal IO/M*.

IO/M* stands for 'input–output/memory*'. When IO/M* is logic 0, it means that the address sent out by the processor is for addressing a memory location. When IO/M* is logic 1, it means that the address sent out by the processor is for addressing an I/O port.

When IO/M* is at logic 1, how to identify whether the address is for an input port or an output port? It is addressing an input port if RD* is at logic 0 and WR* is at logic 1. It is addressing an output port if WR* is at logic 0 and RD* is at logic 1.

To conclude, the action performed by the 8085 for various combinations of RD*, WR*, and IO/M* are as follows.

*RD**	*WR**	*IO/M**	*Action*
0	1	1	8085 reads information from input port
1	0	1	8085 writes information to output port
0	1	0	8085 reads information from memory
1	0	0	8085 writes information to memory

So far, 22 pins of 8085 are explained. The remaining pins are explained in later chapters.

■ 4.3 PROGRAMMER'S VIEW OF 8085: NEED FOR REGISTERS

We have seen that 8085 receives 8-bit information on AD_{7-0} from memory or an input port. When it comes inside the microprocessor, where does it reside inside the processor? For this purpose, there is what is called 'register' to store such information inside. A register is nothing but a group of flip-flops, where each flip-flop can store a bit of information. The size of such a register in 8085 has to be 8 bits, to store 8 bits of information.

It is necessary to have atleast one such register in any brand of microprocessor. The advantages of a register over a memory location are as follows.

- The contents of a register can be accessed much faster by the microprocessor, compared with the contents of a memory location (by 'access' we mean 'read or write').
- Instructions involving register operands will be shorter in length and are executed faster compared with instructions involving memory operands.

However, the disadvantages of a register over a memory location are as follows.

- During an interrupt service subroutine (ISS), the register values will have to be stored in an area of memory called stack. Upon return from the ISS, the register values have to be restored from the stack. This takes away the speed advantage that was mentioned earlier.
- If there are too many registers, they occupy lot of real estate on the chip, thus reducing the real estate available for the control unit and ALU.
- In instructions involving register operands, the number of bits that are free to specify the operation become reduced, thereby reducing the number of possible instructions.

The reader will be in a position to appreciate these advantages and disadvantages much better after a study of a few of the successive chapters.

In view of the earlier discussion, most microprocessors provide only a few registers on the chip. The registers provided on 8085 are A, B, C, D, E, H, and L. These are the general-purpose registers

(GPRs). In addition to these registers there are a few special purpose registers (SPRs). These are discussed later.

4.3.1 MEANING OF PROGRAMMER'S VIEW

A programmer just writes a program and executes it to solve a problem. He is not bothered to know the pins of a processor, or its internal architectural details. To write a program, he needs to know only a little about the processor. For example, he needs to know the registers that are available to the programmer, and the instructions the processor understands to write his program. Thus the programmer's view of a processor, is the simplest description of the processor, without going through many details. It is similar to the driver's view of a car. From a driver's viewpoint there are only a few parts in a car, like steering wheel, brake pedal, etc. He need not even know that there should be an engine! He can still drive a car with so much of ignorance! But in reality a car will have probably a thousand parts or more.

In this chapter, the programmer's view of 8085 is discussed. To keep it simple, the complete programmer's view is not discussed. From the programmer's view, 8085 just consists of a few registers, as shown in Fig. 4.4. They are discussed next.

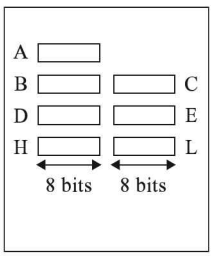


Fig. 4.4
Programmer's view
of 8085 (incomplete)

■ 4.4 ACCUMULATOR OR REGISTER A

Register A is a 8-bit register in 8085. Register A is quite often called as a Accumulator. Among the 8-bit GPRs of 8085, this is the most important. In an arithmetic operation involving two operands, one operand has to be in this register. And the result of the arithmetic operation will be stored or accumulated in this register. This justifies the name 'accumulator' for this register. Similarly, in a logical operation involving two operands, one operand has to be in the accumulator. Also, some operations, like complementing and decimal adjustment, can be performed only on the accumulator.

There are more ways of putting information, as well as taking out information from accumulator compared with the other registers. Technically speaking, there are more ways of addressing the accumulator compared with the other registers.

■ 4.5 REGISTERS B, C, D, E, H, AND L

All these GPRS are 8-bits wide. They are less important than the accumulator. For example, there is no instruction to add contents of D and H registers. Atleast one of the operands has to be in A. Thus to add D and H registers, and to store the result in D register, the following have to be done.

- Move to A register the contents of D register.
- Then add A and H registers. The result will be in A.
- Move this result from A register to D register.

It is possible to use these registers as pairs to store 16-bit information. Only BC, DE, and HL can form register pairs. The instructions of 8085, for example, cannot treat the contents of C and H registers as a single 16-bit quantity. They are always treated as two separate 8-bit quantities.

The information in these register pairs can be interpreted as 16-bit data or 16-bit address of a memory location, depending on the instruction being executed. For example, in the instruction LDAX B, the contents of BC pair are treated as 16-bit memory address. In the instruction DAD B, the contents of BC pair are treated as 16-bit data. These instructions are described in the next chapter.

When they are used as register pairs in an instruction, the left register is understood to have the MS byte and the right register the LS byte. For example, in BC register pair, the content of B register is treated as the MS byte, and the content of C register is treated as the LS byte.

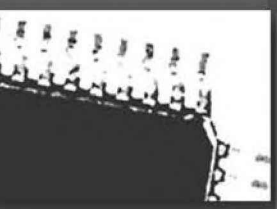
After the names A, B, C, D, E, why it has become H and L, instead of F and G? The reason is as follows.

As far as register pairs are concerned, HL pair is the most important. Although 8085 is a 8-bit processor, the designers have provided for 16-bit addition in the instruction set. In such a 16-bit addition, one of the operands has to be in the HL pair. Also, there are more ways of addressing the HL pair compared with the other register pairs.

In some 8085 instructions there will be a memory operand, indicated as M in an instruction. An example is 'MOV C, M'. This instruction, to be discussed in a later chapter, stands for 'move to C register from memory'. In this instruction, the address of memory location is understood to be specified as the contents of H and L registers. Thus, M in an instruction always stands for contents of a memory location whose address is the contents of H and L. H will have the high part or MS byte of address, and L will have the low part or LS byte of address. So, after A, B, C, D, E it was named as H and L.

1. Briefly describe the evolution of microprocessors.
2. What is functional pin diagram? How does it differ from the pin diagram of an IC?
3. Explain the function of the RD*, WR*, ALE, IO/M* pins of 8085.
4. What is the number of memory locations 8085 can address?
5. The number of address pins in Intel 8086 is 20, and in Intel 80,386 it is 32. How many memory locations are they capable of addressing?
6. Why LS byte of address and data are multiplexed in 8085?

7. Why addresses and data are generally indicated by the user in hexadecimal notation?
8. What is the function of AD_{7-0} pins?
9. What do you understand by programmer's view of a processor?
10. What is a register and its merits/demerits over a memory location?
11. What is the specialty of A register over the other GPRs?
12. Mention the ways B, C, D, E, H, and L registers can be used.
13. What is the specialty of HL pair over the other register pairs?



5 First Assembly Language Program

- Problem statement
  - *Approach methodology*
  - *Program*
  - *Explanation of the program*
- About the microprocessor kit
  - *Major components of a kit*
  - *Functions performed by a microprocessor kit*
  - *Major components of ALS-SDA-85M kit*
  - *Opcode of an instruction mnemonic*
  - *Basics of using the kit*
  - *Entering the program*
  - *Entering the data*
  - *Executing the program without a break*
  - *Executing the program in mode single step*
  - *Using a kit other than ALS-SDA-85M kit*
- Using the microprocessor kit in serial mode
- Questions

Now that we have an idea of the programmer's view of 8085, let us see how it can be used to solve problems for us. To explain this aspect, let us consider a very simple problem.

5.1 PROBLEM STATEMENT

Let us say, we have some 8-bit number in a symbolic memory location X. We want to compute its 2's complement and store the result in symbolic memory location Y, overwriting the previous contents of memory location Y.

5.1.1 Approach Methodology

We need to have memory locations X and Y in the RAM area of the kit. The RAM space in the kit is F800H to FFFFH. So let us arbitrarily choose X as memory location F840H, and Y as memory location F850H.

Let us say, we have in memory location F840H the 8-bit number 12H (i.e. 0001 0010). To find its 2's complement, first of all we have to find its 1's complement, and then add 1 to it. The 1's complement is EDH. Thus the 2's complement becomes EEH. Finally this result is stored in memory location F850H. This is diagrammatically indicated in Fig. 5.1.

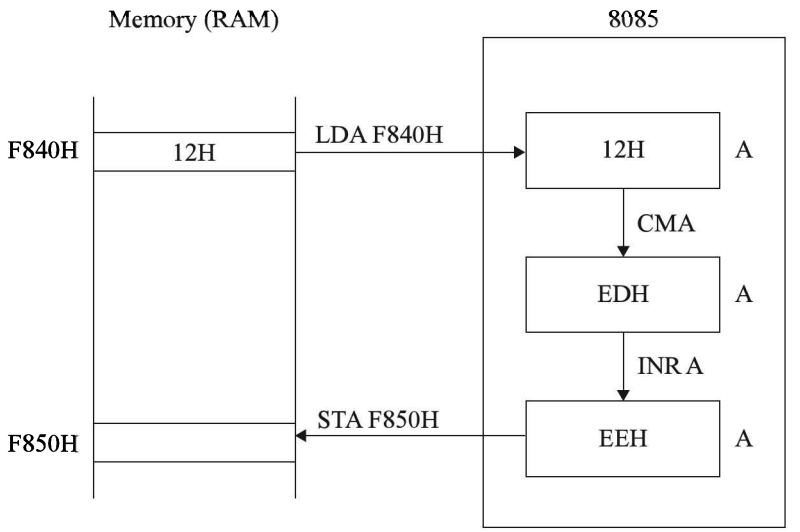


Fig. 5.1
Finding 2's complement of a number

5.1.2 Program

There is nothing like a unique program for solving a problem. However, there will exist a program that is optimal considering one or more of the following factors: ease of developing the program; size of program; and speed of execution of program.

In this book, the programs are written giving highest priority for ease of writing the program, followed by size of program, and finally speed of execution is considered. This is because, speed of execution is most important only in real-time applications, like firing a missile. It is required to minimize size of program, only when the program is too big for the computer's memory. In the present-day world, memory has become very cheap of late. Ease of developing program is very important, because manpower is becoming increasingly expensive.

Program to solve this simple problem is as follows.

```
LDA  F840H    ;  (A)  =  12H
CMA           ;  (A)  =  EDH
INR  A        ;  (A)  =  EEH
STA  F850H    ;  (F850)  =  EEH
HLT
```

5.1.3 Explanation of the Program

Explanation for each of the lines of the program is given as follows.

Line 1—LDA F840H: Finding the 1's complement is done by changing 0s to 1s and 1s to 0s. This is a logical invert or negate operation. In a microcomputer the logic unit of the ALU can only perform this operation. The ALU is inside the microprocessor. So, first of all the number from memory location F840H has to be brought to the microprocessor. But then, which register in 8085 should receive it? As described earlier, Accumulator is the most important 8-bit register, and so it is desired to receive it in Accumulator. This is achieved by executing the instruction 'LDA F840H'.

Here, LDA stands for '**LoaD** the **A**ccumulator from memory location'. Just LDA is meaningless. LDA must be followed by a 16-bit memory address to make sense. We say, 'LDA' is the mnemonic for 'load the accumulator from memory location'. Such mnemonics are designed to help us memorize the actual expansion.

Notice that the first line, as well as other lines of the program, have some more information written after a semicolon. Whatever follows a semicolon is to be treated as a comment only. It is not part of the instruction. It is optional. However, it makes the program easier to understand.

Thus, in the first line '(A) = 12H' is to be treated as a comment. In this book '()' stands for 'contents of'. Thus, '(A) = 12H' is to be read as 'contents of A becomes 12H'. In other words, LDA F840H results in A register content becoming 12H.

Line 2—CMA: Finding the 1's complement of the number in Accumulator is achieved by executing the CMA instruction. Here, CMA is the mnemonic for '**CoM**plement the **A**ccumulator'. The result of this operation will again be stored in the accumulator. Thus, Accumulator value changes to EDH, as indicated in the comment part of this line. In 8085, only Accumulator contents can be complemented. In fact, that was the reason for bringing the number from memory location F840H to Accumulator, and not any other register.

Line 3—INR A: Adding 1 to the number in the accumulator is achieved by executing the INR A instruction. Here, INR A is the mnemonic for '**INcR**ement the **A**ccumulator'. The result of this operation will again be stored in the Accumulator. Thus, Accumulator value changes to EEH, as indicated in the comment part of this line. Thus, we now have in the Accumulator, the 2's complement of the number at memory location F840H.

Line 4—STA F850H: This instruction stores the Accumulator contents in memory location F850H. Here, STA is the mnemonic for '**ST**ore the **A**ccumulator contents in memory location'. Just STA is meaningless. STA must be followed by a 16-bit memory address, like F850H, to make sense. Thus, contents of memory location F850H changes to EEH, as indicated in the comment part of this line. Now the objective of the program is achieved.

Line 5—HLT: 'HLT' is the mnemonic for '**H**alt the processor'. Thus, when this instruction is executed, the processor halts. The reader may be wondering as to why this instruction is necessary at all. The reader may think that once the job is done the processor has to stop without being told so. But, a microprocessor keeps on executing instruction after instruction, unless asked to halt! It is very much unlike a human being, who unless ordered to work, is always whiling away his time!

■ 5.2 ABOUT THE MICROPROCESSOR KIT

To execute the previous program we need a microcomputer. A microprocessor kit is a tool to develop and test user programs. It is quite often called 'μP kit' or simply 'kit'.

5.2.1 Major Components of a Kit

As it is basically a microcomputer, it will have the following.

- 8085 as the microprocessor,
- EPROM to store the monitor program, also called control program,
- Keyboard using which the user can enter his program and data,
- RAM to store user program, data, as well as results,
- Seven-segment LED display, to indicate memory address and contents,
- I/O ports to communicate with peripheral devices, etc.

5.2.2 Functions Performed by a Microprocessor Kit

A kit has in-built features to allow a user to perform the following.

- Enter program and data using the keyboard;
- Execute the program without any break;
- Run the program in single step (useful for detecting errors);
- Examine, and if needed modify, register values;
- Examine, and if needed modify, memory contents;
- Examine, and if needed modify, next/previous memory contents;
- Perform insertion/deletion of bytes in the user program;
- Perform movement of a block of information, etc.

The above features are provided by the monitor program stored in the EPROM. In addition to the above mentioned features, the monitor program provides a number of utility routines also. These routines help in easy program development. The utilities generally provided on the EPROM are: time delay routine; routine to display information on the data and address fields; routine to read from the keyboard, etc.

When Intel developed 8085 microprocessor, they also released SDK-85 (system design kit for 8085). This was the first 8085 microprocessor kit. Subsequently, a number of system integrators have come out with their versions of the 8085 kit. These later versions are superior in their features compared with the SDK-85. In this book, for the purposes of explanation, we have chosen ALS-SDA-85M trainer kit developed by M/s Advanced Electronic Systems, based in Bangalore. Henceforth, by 'kit' we mean ALS-SDA-85M trainer kit.

5.2.3 Major Components of ALS-SDA-85M Kit

Major components of ALS-SDA-85M kit are the following.

- 8085 CPU operating at 3 MHz (approx.),
- 27128 EPROM (capacity 16K bytes, address range 0000H-3FFFH),

- 6116 RAM (capacity 2K bytes, address range F800H-FFFFH),
- Two 8255s (provides 48 I/O lines),
- Keyboard having 28 keys (0 to F, SUBST MEM, NEXT, PREV, etc.),
- Six seven-segment LEDs (four for address field, two for data field),
- 8279 (used for interfacing the keyboard and seven-segment displays),
- 8253 (provides three 16-bit timers),
- 8251 (provides serial interface to connect to a host computer), etc.

The component layout diagram of the ALS kit is shown in Fig. 5.2. The working of all these ICs, and the kit will be discussed in detail in later chapters.

5.2.4 Opcode of an Instruction Mnemonic

First of all, notice that to enter an instruction like 'CMA', we do not have the 'M' key, or to enter an instruction like 'HLT', there is no 'T' key. Then, how to enter such instructions? It was explained earlier that a digital computer understands only 0s and 1s. So, we have to input an instruction to the computer using only 0s and 1s. Thus, instead of typing the mnemonic 'CMA', we have to enter '0010 1111', which is the operation code for the mnemonic CMA. An operation code is also called 'opcode' for short.

Entering 8 bits like this is very tedious. So the kit is provided with a hex keyboard, using which we are allowed to input in hexadecimal. Thus instead of typing '0010 1111', we enter only the two characters '2F' using the keyboard. However, it should be borne in mind that what is stored inside the computer is the binary value 0010 1111.

Similarly, every instruction mnemonic has a corresponding 8-bit opcode, which is entered using the keyboard as two-digit hexadecimal value. The programmer is not required to memorize the opcodes for the various instructions. He is provided with a chart indicating the opcodes for the various instructions. Fig. 5.3 provides the opcodes in hexadecimal for the instruction mnemonics, arranged in alphabetical order.

Instruction length in 8085 can be 1 byte, 2 bytes, or even 3 bytes, although the opcode is always 1-byte long. For example, CMA is only 1-byte long, whereas LDA F840H is 3-bytes long. In LDA F840H, the opcode for LDA is 1-byte long and F840H occupies another 2 bytes. With this information, now we are ready to use the kit.

5.2.5 Basics of Using the Kit

Do we store our program in EPROM or RAM? The advantage of storing it in EPROM would be that the program will not be lost, when power supply fails. But the disadvantage is that we have to use an EPROM programmer to enter the program. If RAM is used, the program can be easily entered, but the disadvantage is that it will be lost if power supply fails. But we decide to store it in RAM for the following reasons.

1. It is easy to enter it in RAM.
2. It is not required to have this program in memory forever or for a long period. We do not mind losing the program once the result is obtained.

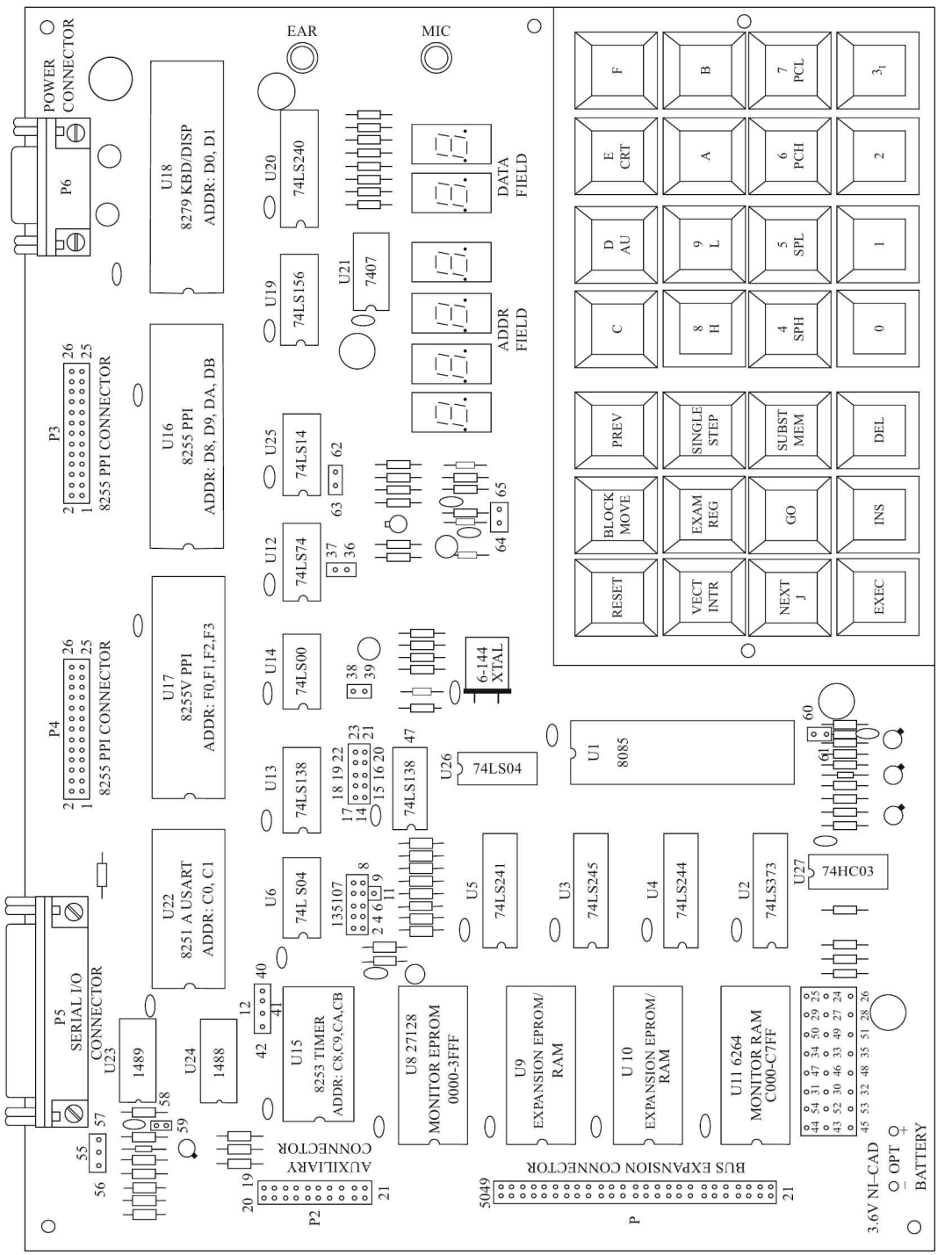


Fig. 5.2 Component layout of ALS-SDA-85M trainer kit

Hex	Mnemonic		Hex	Mnemonic		Hex	Mnemonic		Hex	Mnemonic	
CE	ACI	8-Bit	2B	DCX	H	52	MOV	D,D	E5	PUSH	H
8F	ADC	A	3B	DCX	SP	53	MOV	D,E	F5	PUSH	PSW
88	ADC	B	F3	DI		54	MOV	D,H	17	RAL	
89	ADC	C	FB	EI		55	MOV	D,L	IF	RAR	
8A	ADC	D	76	HLT		56	MOV	D,M	D8	RC	
8B	ADC	E	DB	IN	8-Bit	5F	MOV	E,A	C9	RET	
8C	ADC	H	3C	INR	A	58	MOV	E,B	20	RIM	
8D	ADC	L	04	INR	B	59	MOV	E,C	07	RLC	
8E	ADC	M	OC	INR	C	5A	MOV	E,D	F8	RM	
87	ADD	A	14	INR	D	5B	MOV	E,E	D0	RNC	
80	ADD	B	1C	INR	E	5C	MOV	E,H	C0	RNZ	
81	ADD	C	24	INR	H	5D	MOV	E,L	F0	RP	
82	ADD	D	2C	INR	L	5E	MOV	E,M	E8	RPE	
83	ADD	E	34	INR	M	67	MOV	H,A	E0	RPO	
84	ADD	H	03	INX	B	60	MOV	H,B	0F	RRC	
85	ADD	L	13	INX	D	61	MOV	H,C	C7	RST	0
86	ADD	M	23	INX	H	62	MOV	H,D	CF	RST	1
C6	ADI	8-Bit	33	INX	SP	63	MOV	H,E	D7	RST	2
A7	ANA	A	DA	JC	16-Bit	64	MOV	H,H	DF	RST	3
A0	ANA	B	FA	JM	16-Bit	65	MOV	H,L	E7	RST	4
A1	ANA	C	C3	JMP	16-Bit	66	MOV	H,M	EF	RST	5
A2	ANA	D	D2	JNC	16-Bit	6F	MOV	L,A	F7	RST	6
A3	ANA	E	C2	JNZ	16-Bit	68	MOV	L,B	FF	RST	7
A4	ANA	H	F2	JP	16-Bit	69	MOV	L,C	C8	RZ	
A5	ANA	L	EA	JPE	16-Bit	6A	MOV	L,D	9F	SBB	A
A6	ANA	M	E2	JPO	16-Bit	6B	MOV	L,E	98	SBB	B
E6	ANI	8-Bit	CA	JZ	16-Bit	6C	MOV	L,H	99	SBB	C
CD	CALL	16-Bit	3A	LDA	16-Bit	6D	MOV	L,L	9A	SBB	D
DC	CC	16-Bit	0A	LDAX	B	6E	MOV	L,M	9B	SBB	E
FC	CM	16-Bit	1A	LDAX	D	77	MOV	M,A	9C	SBB	H
2F	CMA		2A	LHLD	16-Bit	70	MOV	M,B	9D	SBB	L
3F	CMC		01	LXI	B, 16-Bit	71	MOV	M,C	9E	SBB	M
BF	CMP	A	11	LXI	D, 16-Bit	72	MOV	M,D	DE	SBI	8-Bit
B8	CMP	B	21	LXI	H, 16-Bit	73	MOV	M,E	22	SHLD	16-Bit
B9	CMP	C	31	LXI	SP, 16-Bit	74	MOV	M,H	30	SIM	
BA	CMP	D	7F	MOV	A,A	75	MOV	M,L	F9	SPHL	
BB	CMP	E	78	MOV	A,B	3E	MVI	A, 8-Bit	32	STA	16-Bit
BC	CMP	H	79	MOV	A,C	06	MVI	B, 8-Bit	02	STAX	B
BD	CMP	L	7A	MOV	A,D	0E	MVI	C, 8-Bit	12	STAX	D
BE	CMP	M	7B	MOV	A,E	16	MVI	D, 8-Bit	37	STC	
D4	CNC	16-Bit	7C	MOV	A,H	1E	MVI	E, 8-Bit	97	SUB	A
C4	CNZ	16-Bit	7D	MOV	A,L	26	MVI	H, 8-Bit	90	SUB	B
F4	CP	16-Bit	7E	MOV	A,M	2E	MVI	L, 8-Bit	91	SUB	C
EC	CPE	16-Bit	47	MOV	B,A	36	MVI	M, 8-Bit	92	SUB	D
FE	CPI	8-Bit	40	MOV	B,B	00	NOP		93	SUB	E
E4	CPO	16-Bit	41	MOV	B,C	B7	ORA	A	94	SUB	H
CC	CZ	16-Bit	42	MOV	B,D	B0	ORA	B	95	SUB	L
27	DAA		43	MOV	B,E	B1	ORA	C	96	SUB	M
09	DAD	B	44	MOV	B,H	B2	ORA	D	D6	SUI	8-Bit
19	DAD	D	45	MOV	B,L	B3	ORA	E	EB	XCHG	
29	DAD	H	46	MOV	B,M	B4	ORA	H	AF	XRA	A
39	DAD	SP	4F	MOV	C,A	B5	ORA	L	A8	XRA	B
3D	DCR	A	48	MOV	C,B	B6	ORA	M	A9	XRA	C
05	DCR	B	49	MOV	C,C	F6	ORI	8-Bit	AA	XRA	D
0D	DCR	C	4A	MOV	C,D	D3	OUT	8-Bit	AB	XRA	E
15	DCR	D	4B	MOV	C,E	E9	PCHL		AC	XRA	H
1D	DCR	E	4C	MOV	C,H	C1	POP	B	AD	XRA	L
25	DCR	H	4D	MOV	C,L	D1	POP	D	AE	XRA	M
2D	DCR	L	4E	MOV	C,M	E1	POP	H	EE	XRI	8-Bit
35	DCR	M	57	MOV	D,A	F1	POP	PSW	E3	XTHL	
0B	DCX	B	50	MOV	D,B	C5	PUSH	B			
1B	DCX	D	51	MOV	D,C	D5	PUSH	D			

Fig. 5.3 Opcode chart for the instruction mnemonics

In ALS-SDA-85M, user program can be stored anywhere in the RAM address range of F800H to FFA0H. Memory locations FFA1H to FFFFH in the RAM are used by the monitor program for storing system variables. As such, the user should avoid this range for storing his program, data, or results.

To enter information in memory, the user has to go through the following steps.

Connecting the power supply: The kit uses power supply module 85M-OPT-04, which provides +5 V dc at 1 A, and ±12 V dc at 0.1 A. Connect power supply to the kit using the cable that is terminated with a nine-pin D-type connector at the kit end. Only two wires are required to be connected at this point. Generally orange, blue, or a white coloured wire is to be connected to the +5 V dc terminal and black coloured wire is to be connected to the ground terminal, marked as 'COM' (for common) on the power supply unit. The other wires are to be left open at this stage. They are needed to connect ±12 V dc to the kit, when it is desired to connect the kit to a host computer.

Sign-on message: Switch on the power supply. If the kit is working fine, it will display '-SdA 85' on the six seven-segment LEDs. The '-' in the beginning indicates that it is ready to accept user commands. It is generally called the system prompt. '-SdA 85' is the sign on message that is displayed whenever power is switched on, or whenever the 'reset' key is typed.

Checking contents of a memory location: Type the key 'SUBST MEM'. It stands for 'substitute memory'. A dot is displayed at the right edge of the address field. Except for the dot, the address and data fields will be blank. It indicates that the kit expects the user to type a four-hex digit address value. If the user wants to see the contents of F820, he has to type the keys 'F', '8', '2', '0' one after another. The display in the address field after each of the above typing will be '000F.', '00F8.', '0F82.', and finally 'F820.'.

Notice that the '.' is still at the edge of the address field. It means that the user can still change the address value if he desires, by typing the new address. Let us say, we are interested in contents of F820H. Then type 'next' key. Then the display in the data field indicates the contents of memory location F820H. Suppose, memory location F820 is having the value 3E, the display after typing 'next' key will be 'F820 3E.'. The typing needed to check contents of location F820H is indicated in Fig. 5.4. In this Fig. '=' stands for a blank character.

key pressed	**addr field**	**data field**
SUBST MEM	= = = =.	= =
F	0 0 0 F.	= =
8	0 0 F 8.	= =
2	0 F 8 2.	= =
0	F 8 2 0.	= =
NEXT	F 8 2 0	3 E.

Fig. 5.4
Typing needed to check contents of location F820H

Changing the contents of a memory location: Notice that the '.' is now at the edge of the data field. It means that, if the user wants to change the data, he is free to do so. Suppose we want to change the contents of F820H to 47H, then type '4' and '7'. The display in the data field after each of the above typing will be '04.', and finally '47.'. Notice that the '.' is still at the edge of the data field. It means that the user can still change the data value if he desires, by typing the new data.

Let us say, the user decides to change the data to 3A. Then, type '3' and 'A'. The display in the data field after each of the above typing will be '73.', and finally '3A.'. Notice that the '.' is still at the edge of the data field. It means that the user can still change the data value if he desires, by typing the new data. Let us say, the user is not interested in any more change. Then type 'next' key. Only now, the

monitor program stores in memory location F820H the value 3AH. Then the display changes to indicate the contents of the next memory location F821H. For example, if location F821H is having the value BEH, the display will become 'F821 bE'.

If the user wants to change the contents of F821H to 40H, he has to now type '4', '0', and finally 'NEXT'. Then 40H is entered into memory location F821H, and the display indicates the contents of location F822H. If the user wants to change the contents of F822H to F8H, he has to now type 'F', '8', and finally 'NEXT'. Then F8H is entered into memory location F822H, and the display indicates the contents of location F823H. If the user is not interested in entering any more information, he has to type 'EXEC' key, to terminate the process of storing information in memory. Power supply is switched off, when there is nothing else to be done on the kit. But then, the information stored in the RAM will be lost.

The typing required to first change the contents of F820H to 47H, and then to 3AH, contents of F821H to 40H, and contents of F822H to F8H is shown in Fig. 5.5. In this diagram 'd d' in the data field indicates any arbitrary two-digit hex value, which is not of interest to us.

key pressed	**addr field**	**data field**
SUBST MEM	= = = =.	= =
F	0 0 0 F.	= =
8	0 0 F 8.	= =
2	0 F 8 2.	= =
0	F 8 2 0.	= =
NEXT	F 8 2 0	d d.
4	F 8 2 0	0 4.
7	F 8 2 0	4 7.
3	F 8 2 0	7 3.
A	F 8 2 0	3 A.
NEXT	F 8 2 1	d d.
4	F 8 2 1	0 4.
0	F 8 2 1	4 0.
NEXT	F 8 2 2	d d.
F	F 8 2 2	0 F.
8	F 8 2 2	F 8.
NEXT	F 8 2 3	d d.
EXEC	- = = =	= =

Fig. 5.5
Typing needed to modify contents of locations F820H, F821H, and F822H

5.2.6 Entering the Program

As an example, let us store the program in section 5.1.2 starting from F820H. The first instruction of our program, LDA F840H, occupies 3 bytes in memory. Notice that in 8085 programs, a 16-bit memory address like F840H, has to be stored in memory in byte reversal form. In this example, 40, which is the LS byte has to be stored in the lower address F821H, and F8, which is the MS byte has to be stored in the higher address F822H. Thus, LDA F840H has to be stored as 3A 40 F8 in the three locations starting from F820H, as shown in Fig. 5.5. Here 3A is the opcode for LDA. There are microprocessors like Motorola 6800, where 16-bit addresses are not stored in memory in byte reversal form. But somehow, in all Intel processors 16-bit addresses are required to be stored in byte reversal form. This notation of storing addresses in memory is known as little endian notation. The notation used in Motorola processors is known as big endian notation.

The second instruction in the program, CMA, occupies only 1 byte in memory. It is stored as 2F in the memory location F823H. The third instruction in the program, INR A, also occupies only 1 byte in memory. It is stored as 3C in the memory location F824H. The fourth instruction in the program, STA F850H, occupies 3 bytes in memory. It is stored as 32 50 F8 starting from memory locations F825H. Finally, the last instruction in the program, HLT, occupies only 1 byte in memory. It is stored as 76 in the memory location F828H.

This program stored in memory using opcodes is called the machine language program. The machine language program is shown in the following.

Memory address	*Memory contents*	*Instruction*
F820H	3AH	LDA F84OH
F821H	40H	
F822H	F8H	
F823H	2FH	CMA
F824H	3CH	INRA
F825H	32H	
F826H	50H	STA F850H
F827H	F8H	
F828H	76H	HLT

The typing required on the keyboard to enter the program is shown in Fig. 5.6.

```
SUBST MEM
F 8 2 0
NEXT
3 A
NEXT
4 0
NEXT
F 8
NEXT
2 F
NEXT
3 C
NEXT
3 2
NEXT
5 0
NEXT
F 8
NEXT
7 6
NEXT
EXEC
```

Fig. 5.6
Typing required on the keyboard to enter the program

5.2.7 Entering the Data

Now that the program is stored in memory, it is now necessary to store in memory the data on which the program is to operate. As this program takes the data from memory location F840H, we store in this location the data value. If we wish to compute 2's complement of 12H, we store 12H in F840H by typing the keys as shown in Fig. 5.7.

```
SUBST MEM
F 8 4 0
NEXT
1 2
NEXT
EXEC
```

Fig. 5.7
Typing required on the keyboard
to enter the data

5.2.8 Executing the Program without a Break

Now the program and data are in memory. To execute the program, we have to just type the keys as shown in Fig. 5.8. In this diagram 'a a a a' in the address field indicates any arbitrary four-digit hex value, which is not of interest to us. Similarly, 'd d' in the data field indicates any arbitrary two-digit hex value.

key pressed	**addr field**	**data field**
GO	a a a a.	d d
F	0 0 0 F.	= =
8	0 0 F 8.	= =
2	0 F 8 2.	= =
0	F 8 2 0.	= =
EXEC	E = = =	= =

Fig. 5.8
Typing required executing
the program without break

Notice that once the program is executed, it displays 'E'. Here, 'E' stands for 'executed'. Of course, many mistake it to mean 'error'! Then to check the result of execution, we have to just look at the contents of memory location F850H. This is done by typing

```
RESET
SUBST MEM
F 8 5 0
NEXT
```

As the data for the program is 12H, its 2's complement is EEH. So, if we get EE displayed in the data field now, we have got the correct answer.

In this example, we have run the entire program without a break. After this, it is a good practice to check up the program with several different data. But every time we change the data, we have to run the program again, and then only we should check the contents of F850H to see the result.

5.2.9 Executing the Program in Single Step mode

In the previous example, if the result is not EE, it means that the program has not produced correct 2's complement. This will be so, when the program is wrong. To locate the error, the facility provided in a kit is the single-step feature.

Using single-step feature, the programmer can see the contents of memory locations, or registers, after the execution of each instruction in the program. If needed, he can even alter the contents of memory locations or registers, after the execution of each instruction. If the user is not interested in seeing the contents of memory locations or registers, he can simply proceed with the next instruction.

To use this feature, the user should first of all note down the effect of execution of each of the instruction in the comments field. This is called 'tracing' the program execution. For example, he should note that A register content should change to 12H, after the execution of LDA F840H. If after executing LDA F840H, register A has the value 34H, something is wrong with this instruction. Then the user should carefully check the coding of the instruction. A common mistake is coding LDA F840H as '3A F8 40', instead of the correct code, which is '3A 40 F8'. Other possible mistakes could be wrong typing of code for LDA. The user by mistake may type '4A' or '3B', instead of the correct code '3A'.

If the tracing of the program gives wrong answer, it is definite that the program is incorrect. If the tracing of the program gives correct answer, it is likely that the program works correctly for all possible values of input data. The user should verify this by giving different sets of data in a judicious way.

To check the program in single step mode, type the keys as shown in Fig. 5.9 when the system prompt appears in the address field.

In the single-step mode, after checking contents of a register, the other sequential registers can be checked by just typing 'NEXT' every time, finally terminating the command by typing 'EXEC'. For example, if the user is only interested in checking contents of H and L registers at the end of execution of an instruction, he has to type the following.

```
EXAM REG
H      (displays H register contents in the data field)
NEXT   (displays L register contents in the data field)
EXEC   (terminates the 'EXAM REG' command)
```

In this, notice that to display contents of H register the key on which both H and 8 are printed has to be typed. After typing 'EXAM REG', if this key is typed, it is interpreted as H. Similarly, after typing 'SUBST MEM', if this key is typed, it is interpreted as 8. There are some more keys on the keyboard, which have two values engraved.

Similarly, in the single-step mode, after checking contents of a memory location, the other sequential locations can be checked by just typing 'NEXT' every time, finally terminating the command by typing 'EXEC'. For example, if the user is only interested in checking contents of memory locations F850H, F851H, and F852H at the end of execution of an instruction, he has to type the following.

```
SUBST MEM
F 8 5 0  (displays F850 in the address field)
NEXT     (displays contents of F850 in the data field)
NEXT     (displays contents of F851 in the data field)
NEXT     (displays contents of F852 in the data field)
EXEC     (terminates the 'SUBST MEM' command)
```

key pressed	addr field	data field
SINGLE STEP	a a a a.	d d
F	0 0 0 F.	= =
8	0 0 F 8.	= =
2	0 F 8 2.	= =
0	F 8 2 0.	= =
NEXT	F 8 2 0.	3 A
NEXT	F 8 2 3.	2 F
EXEC	- = = =	= =
EXAM REG	= = = =.	= =
A	= = = A	1 2.
EXEC	- = = =	= =
SINGLE STEP	F 8 2 3.	2 F
NEXT	F 8 2 4.	3 C
EXEC	- = = =	= =
EXAM REG	= = = =.	= =
A	= = = A	E d.
EXEC	- = = =	= =
SINGLE STEP	F 8 2 4.	3 C
NEXT	F 8 2 5.	3 2
EXEC	- = = =	= =
EXAM REG	= = = =.	= =
A	= = = A	E E.
EXEC	- = = =	= =
SINGLE STEP	F 8 2 5.	3 2
NEXT	F 8 2 8.	7 6
EXEC	- = = =	= =
SUBST MEM	= = = =.	= =
F	0 0 0 F.	= =
8	0 0 F 8.	= =
5	0 F 8 5.	= =
0	F 8 5 0.	= =
NEXT	F 8 5 0	E E.
EXEC	- = = =	= =
SINGLE STEP	F 8 2 8.	7 6
EXEC	- = = =	= =
RESET	- S d A	8 5

Fig. 5.9 Typing needed to trace the program in single step mode

5.2.10 Using a Kit Other than ALS-SDA-85M

When the user has access to a kit by some other manufacturer, there are likely be to some differences in the features compared with the ALS-SDA-85M kit. The differences are likely to be in the following details.

- EPROM capacity and address range,
- RAM capacity and address range,
- Number of I/O ports and their address,
- Utility routines and their addresses,
- Availability of special peripheral chips and their addresses, and
- Differences in naming of some keys.

The user is required to go through the manual for the kit to get the necessary information.

5.3 USING THE MICROPROCESSOR KIT IN SERIAL MODE

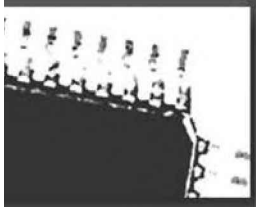
So far we have discussed the use of the kit in the keyboard mode. In this case, the user has to translate the assembly language program into machine code, and enter the program in hexadecimal using the keyboard. This translation is quite monotonous, especially when the program is quite long.

If we have a computer with an ASCII keyboard, the program can be entered into computer memory straightaway as an assembly language program. Then, the assembler program in the computer can translate this assembly language program to machine code, relieving the user from this monotonous task. The machine code can finally be downloaded from the computer to the kit using RS-232C serial link. The user can then execute the downloaded program on the kit.

Also, the user can upload a machine language program from the memory of the kit to the hard disk or the floppy disk on the computer. The hard disk or floppy disk is non-volatile, which means it does not lose the information even after power is switched off and then turned on. Thus the user programs can be stored as disk files on the computer, for downloading later to the kit for execution. However, the cost goes up in serial mode, as a computer is a must in addition to the kit. Also, there must be assembler software on the computer. The circuitry and the monitor software on the kit also is increased.

The actual procedure for using the kit in serial mode is explained in a later chapter.

1. What are the factors to be considered while developing a program?
2. What is a microprocessor kit? Describe the main features of a typical microprocessor kit.
3. What is an opcode? What is the size of an opcode, and an instruction in 8085?
4. What is the need for single stepping through a program? What are the features provided by the single-step function?
5. What is the advantage of using the kit in serial mode? What is the principle of its working?



6 Data Transfer Group of Instructions

- Classification of 8085 instructions
  - *Number of instructions in 8085*
    - Instruction type MVI r, d8
    - Instruction type MOV r1, r2
    - Instruction type MOV r, M
    - Instruction type MOV M, r
    - Instruction type LXI rp, d16
    - Instruction type MVI M, d8
    - Instruction type LDA a16
    - Instruction type STA a16
    - Instruction type XCHG
- Addressing modes of 8085
  - *Register codes*
  - *Immediate addressing mode*
  - *Register addressing mode*
  - *Absolute addressing mode*
  - *Register indirect addressing mode*
  - *Implied addressing mode*
    - Instruction type LDAX rp
    - Instruction type STAX rp
    - Instruction type LHLD a16
    - Instruction type SHLD a16
- Questions

In this chapter a total of 13 instruction types covering 83 instructions are explained. In the previous chapter a very simple program to compute the 2's complement of a number was discussed. The program was as follows.

```
LDA F840H   ;    (A)   = 12H
CMA         ;    (A)   = EDH
INR A       ;    (A)   = EEH
STA F850H   ; (F850)   = EEH
HLT
```


In this program, LDA F840H execution results in movement of information from F840H to Accumulator. Thus it is an example for data transfer group of instructions. CMA execution results in complementing Accumulator contents. It is a logical invert operation. Thus it is an example for logical group of instructions. INR A execution results in incrementing Accumulator contents. It adds 1 to the Accumulator contents. Thus it is an example for arithmetic group of instructions. STA F850H execution results in movement of information from Accumulator to memory location F850H. Thus it is one more example for data transfer group of instructions. HLT execution results in halting the processor. It is an example for machine control group of instructions, as this instruction controls the machine, which is the microcomputer here.

■ 6.1 CLASSIFICATION OF 8085 INSTRUCTIONS

From the previous discussion, we find that 8085 microprocessor executes a variety of instructions, which can be broadly classified as follows.

Description	*No. of opcodes*	*No. of instruction types*
Data transfer instructions	83	13
Arithmetic instructions	62	14
Logical instructions	43	15
Stack instructions	15	9
Branch instructions	36	8
I/O instructions	2	2
Interrupt instructions	5	5
Total	246	66

Of course, in the earlier simple program, we have not come across some groups of instructions, like the branch group. But, in many programs we frequently come across such instructions. To write a program, we need to first of all know all the instructions that the 8085 can execute.

Towards this objective, in this chapter we describe the data transfer group of instructions, and in the successive few chapters we describe the other groups of instructions.

6.1.1 Number of Instructions in 8085

As we have already seen from the previous program, although instruction may be as large as 3 bytes, the opcode is always 1 byte in length. With 8 bits for the opcode, $2^8 = 256$ distinct opcodes are possible. In hexadecimal, the opcodes can be from 00H to FFH. Each opcode corresponds to an instruction. Thus theoretically, 256 instructions are possible in the instruction set of 8085. However, only 246

opcodes are implemented in 8085. They can be discussed under 66 types, which are broadly classified into the six groups listed earlier. Of the 246 opcodes we have:

- 202 opcodes that are 1-byte long,
- 18 opcodes that are 2-byte long, and
- 26 opcodes that are 3-byte long.

In data transfer group, we have 83 opcodes, which will be discussed in this chapter under 13 different types.

■ 6.2 INSTRUCTION TYPE MVI r, d8

MVI is a mnemonic, which stands for '**M**o**V**e **I**mmediate'. This is an instruction to load a register with an 8-bit value. This instruction uses immediate addressing for specifying the data. We will discuss about addressing modes in a later section. In this type, 'd8' stands for any 8-bit data, and 'r' stands for any of the following registers.

r = A, B, C, D, E, H, or L

As 'r' can have any of the seven values, there are seven opcodes for this type of instruction. It occupies 2 bytes in the memory. MVI E, 8DH is an example instruction of this type. It is a 2-byte instruction, with opcode for MVI E using up one byte, and 8DH using up one more byte. Suppose E register content is 45H. When the 8085 executes this instruction, the contents of E register will change to 8DH. This is shown below in an easy-to-understand manner.

	Before	***After***
(E)	45	8D

As summary for the above discussed instruction type, we provide within brackets the size of the instruction, a representative instruction, and the number of opcodes for this type of instruction, as follows.

MVI r, d8 (2 byte; MVI E, 8EH; 7 opcodes)

In fact, from now on, for each instruction this type of summary will be provided.

■ 6.3 INSTRUCTION TYPE MOV r1, r2

MOV is a mnemonic, which stands for '**MOV**e'. This is an instruction to load register r1 with the 8-bit value in register r2. Notice that in 8085 instructions, the first operand specifies the destination, and the second the source. This instruction uses register addressing for specifying the data. In this type, 'r1' and 'r2' stand for any of the following registers.

r1, r2 = A, B, C, D, E, H, or L

As r1 can have any of the seven values, and r2 can have any of the seven values, there are $7 \times 7 = 49$ opcodes for this type of instruction. It occupies only 1 byte in memory. MOV E, H is an example instruction of this type. It is a 1-byte instruction. Suppose E register content is 45H, and H register content is

8DH. When the 8085 executes this instruction, the contents of E register will change to 8DH. This is shown as follows.

	Before	*After*
(E)	45	8D
(H)	8D	8D

Notice that H register content is not altered at all. Although Intel has called it a 'move' instruction, it is in reality a 'copy' instruction.

Another thing to note is that there are instructions like 'MOV D, D'. This instruction moves contents of D register to D register itself. It is of no use to anybody! However, such useless instructions are also provided in the instruction set of 8085, and some very useful instructions that a programmer looks for are not provided!

Summary: MOV r1, r2 (1 byte; MOV E, H; 49 opcodes)

■ 6.4 INSTRUCTION TYPE MOV r, M

This is an instruction to load register r with the 8-bit value in memory location. But from which memory location? The address of the memory location is understood to be provided in HL register pair. This instruction uses register-indirect addressing for specifying the data.

As r can have any of the seven values, there are seven opcodes for this type of instruction. It occupies only 1 byte in memory. MOV E, M is an example instruction of this type. It is a 1-byte instruction. Suppose E register content is 45H, H register content is F8H, and L register content is 50H. Let us say location F850H has the data value 8DH. When the 8085 executes this instruction, the contents of E register will change to 8DH, as shown below.

	Before	*After*
(E)	45	8D
(HL)	F850	F850
(F850)	8D	8D

Henceforth, for simplicity, only values that have changed will be shown in the 'After' column. Thus the simplified diagram for the execution of MOV E, M will be as follows.

	Before	*After*
(E)	45	8D
(HL)	F850	
(F850)	8D	

Notice that there are instructions like 'MOV H, M'. This instruction moves contents of a memory location pointed by HL register pair to H register. Generally, it is of no use to anybody! However, such useless instructions are also provided in the instruction set of 8085.

Summary: MOV r, M (1 byte; MOV E, M; 7 opcodes)

■ 6.5 INSTRUCTION TYPE MOV M, r

This is an instruction to load a memory location with the 8-bit value in register 'r'. But which memory location? The address of the memory location is understood to be provided in HL register pair. This instruction uses register addressing for specifying the data.

As 'r' can have any of the seven values, there are seven opcodes for this type of instruction. It occupies only one byte in memory. MOV M, E is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown with an example below.

	Before	***After***
(E)	45	
(HL)	F850	
(F850)	8D	45

Notice that there are instructions like 'MOV M, H'. This instruction moves contents of register H to memory location pointed by HL register pair . Generally, it is of no use to anybody! However, such useless instructions are also provided in the instruction set of 8085.

Summary: MOV M, r (1 byte; MOV M, E; 7 opcodes)

■ 6.6 INSTRUCTION TYPE LXI rp, d16

So far we have seen that whenever M is used in an instruction mnemonic, we are referring to the contents of a memory location whose address is given in HL register pair. Thus, if MOV E, M is executed, register E receives contents of memory pointed by HL. Suppose we want E to receive contents of memory location FF00H. Then we have to make sure that HL will have FF00H, before MOV E, M is executed. Thus any instruction involving M, like 'MOV E, M', should generally be preceded by instruction/s to load H and L registers with the desired memory address.

How to ensure that HL pair will have the value FF00H? One method is to use the following two instructions: MVI H, FFH and MVI L, 00H.

These two instructions occupy a total of 4 bytes in memory. A simpler alternative is to use the 3-byte instruction 'LXI H, FF00H', of the type 'LXI rp, d16'.

LXI is a mnemonic that stands for '**L**oad e**X**tended register **I**mmediate'. Here, a register pair is termed as extended register. It is an instruction that loads register pair 'rp' with the 16-bit data denoted as d16. This instruction uses immediate addressing for specifying the data. In this type, 'rp' stands for any of the following register pairs.

rp = BC, DE, or HL

As 'rp' can have any of the three values, there are three opcodes for this type of instruction. It occupies 3 bytes in memory. First byte specifies the opcode, and the next two bytes provide the 16-bit data. LXI H, F850H is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(HL)	FABCH	F850H

Notice that when specifying the register pair, we specify only the MS register in the instruction mnemonic. The 'X' in the mnemonic indicates that it is a register pair, and so if the MS register is specified the register pair is identified. Thus, if our interest is to load HL with F850H, we have to write the instruction as 'LXI H, F850H' and not as 'LXI HL, F850H' or 'LXI L, F850H'. LXI H, F850H is stored in memory with F850 stored in byte reversal form as shown below.

Code for LXI H
50
F8

The other instructions of this type are LXI B, d16 and LXI D, d16.

Summary: LXI rp, d16 (3 bytes; LXI H, F850H; 3 opcodes)

■ 6.7 INSTRUCTION TYPE MVI M, d8

This is an instruction to load a memory location pointed by HL pair with an 8-bit value. This instruction uses immediate addressing for specifying the data. It occupies 2 bytes in memory.

MVI M, 8DH is an example instruction of this type. It is a 2-byte instruction, with opcode for MVI M using up one byte, and 8DH using up one more byte. The result of execution of this instruction is shown below with an example.

	Before	***After***
(HL)	F845	
(F845)	AC	8D

Summary: MVI M, d8 (2 bytes; MVI M, 8DH; 1 opcode)

■ 6.8 INSTRUCTION TYPE LDA a16

LDA is a mnemonic that stands for **LoaD** Accumulator contents from memory. This is an instruction to load Accumulator with the contents of a memory location whose 16-bit address is indicated in the instruction as a16. This instruction uses absolute addressing for specifying the data. It occupies 3 bytes in memory. First byte specifies the opcode, and the successive 2 bytes provide the 16-bit address.

LDA F850H is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(F850)	BCH	
(A)	12H	BCH

LDA F850H is stored in memory with F850 stored in byte reversal form as shown below.

Code for LDA
50
F8

There are no instructions in 8085 like LDB a16, LDC a16, etc. As was stated earlier, Accumulator is the most important 8-bit register, whose contents can be loaded in more ways than any other 8-bit register.

Summary: LDA a16 (3 bytes; LDA F850H; 1 opcode)

■ 6.9 INSTRUCTION TYPE STA a16

STA is a mnemonic that stands for **ST**ore **A**ccumulator contents in memory. This is an instruction to store Accumulator contents in a memory location whose 16-bit address is indicated in the instruction as a16. This instruction uses absolute addressing for specifying the destination. It occupies 3 bytes in memory. First byte specifies the opcode, and the successive 2 bytes provide the 16-bit address.

STA F850H is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(A)	12H	
(F850)	BCH	12H

STA F850H is stored in memory with F850 stored in byte reversal form as shown below.

Code for STA
50
F8

There are no instructions in 8085 like STB a16, STD a16, etc. As was stated earlier, Accumulator is the most important 8-bit register, whose contents can be stored in memory in more ways than any other 8-bit register.

Summary: STA a16 (3 bytes; STA F850H; 1 opcode)

■ 6.10 INSTRUCTION TYPE XCHG

XCHG is a mnemonic, which stands for e**XCH**an**G**e. This is an instruction to exchange contents of HL pair with DE pair. This instruction uses implied addressing. It occupies only 1 byte in memory. The result of execution of this instruction is shown below with an example.

	Before	***After***
(HL)	1234H	5678H
(DE)	5678H	1234H

Using XCHG instruction, only HL and DE contents can be exchanged. There are no instructions in 8085 to exchange contents of HL and BC or to exchange contents of DE and BC.

Summary: XCHG (1 byte; XCHG; 1 opcode)

■ 6.11 ADDRESSING MODES OF 8085

Shown in the following are the sizes of a few of the instruction types.

MOV r1, r2 1 byte
MOV r, M 1 byte
MVI r, d8 2 bytes

Let us see the reason for this before we discuss the addressing modes.

Register	*Register code*
B	000
C	001
D	010
E	011
H	100
L	101
M	110
A	111

Fig. 6.1
Three-bit register codes for 8085 registers

6.11.1 REGISTER CODES

The 8085 sends out 16-bit address to access one of the 2^{16} = 64K locations. But, for convenience, we indicate memory address using four hexadecimal digits. Similarly, for convenience, we indicate registers as A, B, C, etc. But in reality, the microprocessor has to specify these registers using 0s and 1s only. With only seven registers in 8085, 3 bits are enough to specify a register. The 3-bit register codes for the registers of 8085 are tabulated in Fig. 6.1.

With 3-bit register code, eight registers can be specified. But 8085 has only seven registers. So notice that register code 110 does not specify any register.

Fortunately, the user is not required to memorize these register codes. It is enough if the user realizes that 3 bits are needed to specify a register.

Opcode for MOV E, H: Now let us see how Intel has arrived at the opcode for MOV E, H. Six bits are needed to specify the two registers E and H. Still 2 bits are left in a byte. These 2 bits indicate the code for 'MOV'. The template chosen by Intel for instruction type MOV r1, r2 is shown in the following. Intel chose 0 1 as the code for MOV.

0 1 r1 code r2 code

From the previous register code table, the opcode for MOV E, H will be

0 1 0 1 1 1 0 0 = 5CH

This can be verified from the opcode chart given in the previous chapter. Similarly, the code for MOV A, B will be: 0 1 1 1 1 0 0 0, which is 78H.

Opcode for MOV E, M: Now let us see how Intel has arrived at the opcode for MOV E, M. How to specify M? Intel solved this problem by choosing 110, which was not used to specify any register, as the code for M. Thus, although M stands for the contents of a memory location pointed by HL, it has 110 as the 'register code'. Six bits are needed to specify E and M. The remaining 2 bits in the byte indicate the code for 'MOV'. From the previous register code table, the opcode for MOV E, M will be

0 1 0 1 1 1 1 0 = 5EH

This can be verified from the opcode chart given in the previous chapter. Similarly, the code for MOV A, M will be: 0 1 1 1 1 1 0, which is 7EH.

Opcode for MVI E, 34H: Now let us see how Intel has arrived at the opcode for MVI E, 34H. It will be a 2-byte instruction, with 34H as the second byte. In the first byte, 3 bits are needed to specify the registers E. Still 5 bits are left in the first byte. These 5 bits indicate the code for 'MVI'. The template chosen by Intel for instruction type MVI r, d8 is shown as follows.

0 0 r code 1 1 0

From the previous register code table, the opcode for MVI E will be

0 0 0 1 1 1 1 0 = 1EH

This can be verified from the opcode chart given in the previous chapter. Similarly, the code for MVI M will be: 0 0 1 1 0 1 1 0, which is 36H.

Need for addressing modes: A user would like to have the ability to access data in different ways for reasons of convenience. These different ways of accessing data is called the 'addressing modes'. It is something similar to the way a teacher in a classroom would refer to a particular student in one of the following ways.

- By mentioning the name of the student;
- By indicating the position of his seat in the class;
- By indicating the features of the student, etc.

Consider the execution of the following four instructions. In each of these instructions we move 25H to the accumulator, using different addressing modes.

1. MVI A, 25H

	Before	***After***
(A)	12H	25H

2. MOV A, B

	Before	***After***
(B)	25H	
(A)	12H	25H

3. LDA F850H

	Before	***After***
(F850)	25H	
(A)	12H	25H

4. MOV A, M

	Before	***After***
(HL)	F850H	
(F850)	25H	
(A)	12H	25H

6.11.2 Immediate Addressing Mode

In the previous examples, only MVI A, 25H clearly indicates that 25H has to be moved to the A register. This instruction occupies 2 bytes in memory as follows.

Opcode for MVI A
25H

As the data to be moved is immediately after the opcode in the instruction, this kind of addressing is called immediate addressing mode.

6.11.3 Register Addressing Mode

In the previous examples, MOV A, B indicates that contents of the B register have to be moved to the A register. It does not directly say that 25H has to be moved. The instruction only provides the address of the data. In this case, the address is register B, provided in the opcode using 3 bits. This kind of addressing where the data is specified in a register is called Register addressing mode.

6.11.4 Absolute Addressing Mode

In the previous examples, LDA F850H indicates that contents of the memory location F850H have to be moved to the A register. It does not directly say that 25H has to be moved. The instruction only provides the address of the data. In this case, the memory address is F850H, provided in the instruction using 2 bytes. This kind of addressing where the data is specified in a memory location is called absolute addressing mode. It is called absolute addressing because a full 16-bit memory address is given as part of the instruction, in comparison with register addressing where only a 3-bit register address is provided as part of the instruction.

This type of addressing is sometimes called direct addressing.

6.11.5 Register Indirect Addressing Mode

In the previous examples, MOV A, M indicates that contents of the memory location pointed by HL register pair has to be moved to A register. It does not directly say that 25H has to be moved.

If HL content is F850H, we are not moving F850H to Accumulator. We have to take F850H as the memory address, and so move contents of F850H. This is an indirect way of using register pair HL. Hence, this instruction is said to use Register indirect addressing.

6.11.6 Implied Addressing Mode

Consider the instruction XCHG. It exchanges the contents of HL pair with DE pair. But the instruction mnemonic does not explicitly specify that HL and DE are to be exchanged. In other words, we do not write the instruction as 'XCHG HL, DE'. It is just implied that we have to exchange HL and DE. So this type of addressing is called implied addressing mode. It is also called implicit addressing mode, sometimes.

These are the only addressing modes available in 8085. Some other important addressing modes, like Indexed addressing, are not provided in 8085.

■ 6.12 INSTRUCTION TYPE LDAX rp

LDAX is a mnemonic that stands for **LoaD** Accumulator from memory pointed by e**X**tended register denoted as 'rp'. This instruction uses register indirect addressing for specifying the data. It occupies only 1 byte in memory.

LDAX B is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(BC)	F2BCH	
(F2BC)	DCH	
(A)	12H	DCH

Only other instruction of this type is LDAX D. Note that LDAX H is not provided in 8085. This is because, LDAX H is the same as MOV A, M in its function.

Also note that there are no instructions in 8085 like LDBX rp, LDCX rp, etc. As was stated earlier, Accumulator is the most important 8-bit register, whose contents can be loaded in more ways than any other 8-bit register.

Summary: LDAX rp (1 byte; LDAX B; 2 opcodes)

■ 6.13 INSTRUCTION TYPE STAX rp

STAX is a mnemonic that stands for **ST**ore **A**ccumulator contents in memory pointed by e**X**tended register denoted as 'rp'. This instruction uses register indirect addressing for specifying the destination. It occupies only 1 byte in memory.

STAX B is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(BC)	F2BCH	
(A)	12H	
(F2BC)	DCH	12H

Only other instruction of this type is STAX D. Note that STAX H is not provided in 8085. This is because, STAX H is the same as MOV M, A in its function.

Also note that there are no instructions in 8085 like STBX rp, STCX rp, etc. As was stated earlier, Accumulator is the most important 8-bit register, whose contents can be stored in memory in more ways than any other 8-bit register.

Summary: STAX rp (1 byte; STAX B; 2 opcodes)

■ 6.14 INSTRUCTION TYPE LHLD a16

LHLD is a mnemonic that stands for **L**oad **HL** pair using **D**irect addressing from memory location whose 16-bit address is denoted as a16. As HL pair has to be loaded, the data comes from two consecutive locations starting at the address a16. This instruction uses absolute addressing for specifying the data. It occupies 3 bytes in memory.

LHLD F2BCH is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(F2BC)	DCH	
(F2BD)	ABH	
(H)	12H	ABH
(L)	34H	DCH

LHLD F2BCH is stored in memory with F2BC stored in byte reversal form as shown below.

Code for LHLD
BC
F2

Note that there are no instructions in 8085 like LBCD a16 and LDED a16. As was stated earlier, HL pair is the most important register pair, whose contents can be loaded in more ways than any other register pair.

Summary: LHLD a16 (3 bytes; LHLD F2BCH; 1 opcode)

■ 6.15 INSTRUCTION TYPE SHLD a16

SHLD is a mnemonic, which stands for **S**tore **HL**pair using **D**irect addressing in memory location whose 16-bit address is denoted as a16. As HL pair has to be stored, it has to be stored in two consecutive locations starting at the address a16. This instruction uses absolute addressing for specifying the destination. It occupies 3 bytes in memory.

SHLD F2BCH is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is pictorially shown below with an example.

	Before	*After*
(H)	12H	
(L)	34H	
(F2BC)	DCH	34H
(F2BD)	ABH	12H

SHLD F2BCH is stored in memory with F2BC stored in byte reversal form as shown below.

Code for SHLD
BC
F2

Note that there are no instructions in 8085 like SBCD a16 and SDED a16. As was stated earlier, HL pair is the most important register pair, whose contents can be stored in memory in more ways than any other register pair.

Summary: SHLD a16 (3 bytes; SHLD F2BCH; 1 opcode)

1. Classify the different types of instructions available in 8085, giving an example for each type.
2. How many different instructions are implemented in 8085?
3. Distinguish between the following pairs of instructions:
   a. LXI H, 1234H and LHLD 1234H;
   b. LDA F900H and STA F900H;
   c. MVI M, 8DH and LXI H, 008DH;
   d. LHLD FA00H and SHLD FA00H.
4. What do you mean by addressing mode? Explain the different types of addressing modes available in 8085, with an example for each.
5. Using the register code table, derive opcodes for the following instructions:
   i) MOV B, C, ii) MOV M, D, iii) MOV A, M.

7 Arithmetic Group of Instructions

- Instructions to perform addition
  - *Instruction type ADD R*
    - *Flags register*
  - *Instruction type ADI d8*
  - *Instruction type INR R*
  - *Instruction type ADC R*
  - *Instruction type ACI d8*
- Instructions to perform subtraction
  - *Instruction type SUB R*
  - *Instruction type SUI d8*
  - *Instruction type DCR R*
  - *Instruction type SBB R*
  - *Instruction type SBI d8*
- Instruction type INX rp
- Instruction type DCX rp
- Instruction type DAD rp
- Decimal addition in 8085
  - *BCD numbers*
  - *DAA instruction*
- Questions

A total of 14 instruction types covering 62 instructions will be explained in this chapter. The various instruction types to perform addition and subtraction operations are dealt with in detail in this chapter. Furthermore, the instruction types INX rp, DCX rp, and DAD rp are also discussed briefly in the latter portion of the chapter.

7.1 INSTRUCTIONS TO PERFORM ADDITION

Intel 8085 is a primitive microprocessor. In its arithmetic group of instructions, it has only add and subtract instructions. It does not have instructions to multiply or divide numbers. In this section we discuss the instructions to perform addition.

In the addition of two numbers, 8085 imposes the restriction that one of the operands must be in the Accumulator. The other operand can be one of the following.

—Contents of an 8-bit register,
—Contents of memory location pointed by HL pair,
—Eight-bit immediate data.

7.1.1 Instruction Type ADD R

ADD is a mnemonic that stands for 'Add contents of R to accumulator'. In this case, 'R' stands for any of the following registers or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to add contents of R and accumulator. The result of the addition will be stored in the accumulator.

As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory.

'ADD E' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(E)	F3H	
(A)	45H	38H

Summary: ADD R (1 byte; ADD E; 8 opcodes)

7.1.2 Flags Register

In the previous example, the result of addition of 45H and F3H turns out to be 38H, which is less than the original value of 45H present in the accumulator! Obviously, the user will not be happy with such state of affairs. The correct answer is 38H with carry of 1. This carry information is stored in a special 8-bit register called the flags register, F for short. A flag can be in the hoisted state, or it is not hoisted at all! Thus a flag can be represented by 1 bit of information. Thus the flags register can have a total of eight flags. But only five flags are implemented in 8085. They are:

- Carry flag ('Cy'),
- Auxiliary carry flag (AC),
- Sign flag (S),
- Parity flag (P), and
- Zero flag (Z).

Their positions in the flags register is shown in Fig. 7.1. The positions marked 'x' are don't care bits in the flags register. The user is not required to memorize the positions of these flags in the flags register.

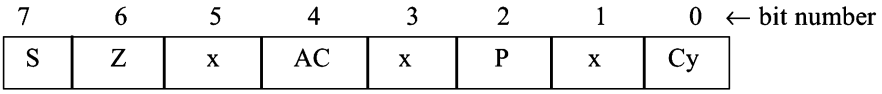


Fig. 7.1 Flags register

Thus the programmer's view of 8085 contains the flags register also, as shown in Fig. 7.2.

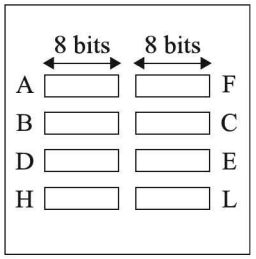


Fig. 7.2
Programmer's view
of 8085 (incomplete)

These individual flags are either hoisted (set to 1), or not hoisted (reset to 0) depending on the result of execution of an arithmetic or logical instruction. But in a few arithmetic and logical instructions, some or none of these flags are affected.

However, in any data transfer instruction, none of the flags are affected. Hence the topic of flags register was not discussed at all while explaining data transfer instructions in the previous chapter.

Carry flag (Cy): Notice that in the addition of any two 8-bit numbers, the carry generated can be either 0 or 1. Thus to store the carry information 1-bit storage is enough. The Cy flag is stored in the LS bit position in the flags register. Instructions that use the Cy flag are widely used in the user programs.

Example 1: In the addition of 45H and F3H, the result is 38H with Cy flag = 1, as shown below.

```
Cy AC
  10
    4 5H
  + F 3H
  ---------
    3 8H = 0011 1000
```

Example 2: In the addition of 85H and 1EH, the result is A3H with Cy = 0, as shown below.

```
Cy AC
  01
    8 5H
  + 1 EH
  ---------
    A 3H = 1010 0011
```

Auxiliary carry flag (AC): In the addition of any two 8-bit (2-hex digit) numbers, a carry may be generated when we add the LS hex digits of the two numbers. Such a carry is called intermediate carry, half carry, or auxiliary carry (AC). Intel prefers to call it AC. In Ex. 1, AC is not generated. In Ex. 2, AC is generated.

As this is only an intermediate carry, we are not interested in storing this information. But 8085 still stores this AC information in bit position 4 of the flags register. The result of execution of DAA instruction, to be described later, is affected by the status of this flag. However, 8085 instruction set does not provide any instruction, which explicitly uses the AC flag.

Sign flag (S): The S flag is set to 1, if the result of an arithmetic operation is negative, indicated by MS bit of 8-bit result being 1. It is reset to 0 if the result is positive, indicated by MS bit of 8-bit result being 0.

Thus, the value of S flag is essentially the value of the MS bit of the 8-bit result. In Ex. 1, as the 8-bit result is 38H = 0 011 1000, the sign flag is reset to 0. Note that we are not considering here the 9-bit result including the carry, to decide the S flag value. In Ex. 2, as the 8-bit result is A3H = 1 010 0011, the sign flag is set to 1.

If we are working with unsigned numbers, we simply ignore the S flag. For example, if we are treating 85H and 1EH as unsigned numbers, their sum will be the unsigned number A3H. In this case, S flag becomes 1, but we do not care for the value of the S flag.

Instructions that use the S flag are quite often used in the user programs.

Parity flag (P): The P flag is set to 1, if the 8-bit result of an arithmetic operation has an even number of 1's in it. If there are odd number of 1's in the 8-bit result, the P flag is reset to 0.

In Ex. 1, as the 8-bit result 38H = 0011 1000 has three numbers of 1's (an odd number), the parity flag is reset to 0. In Ex. 2, as the 8-bit result A3H = 1010 0011 has four numbers of 1's (an even number), the parity flag is set to 1.

As the user does not really care for the number of 1's present in the result after an arithmetic operation, this flag is not of much use practically.

Zero flag (Z): The Z flag is set to 1, if the 8-bit result of an arithmetic operation is 00H. If the 8-bit result is not equal to 00H, the Z flag is reset to 0. Thus the Z flag is hoisted to indicate that the result is 0.

In Ex. 1, as the 8-bit result is 38H and is non-zero, the Z flag is reset to 0. Also in Ex. 2, as the 8-bit result is A3H, the Z flag is reset to 0. Instructions that use the Z flag are widely used in the user programs.

7.1.3 Instruction Type ADI d8

ADI is a mnemonic, which stands for '**AD**d **I**mmediate to Accumulator' and 'd8' stands for any 8-bit data. This instruction is used to add 8-bit immediate data to the Accumulator. The result of addition will be stored in the accumulator. It occupies 2-bytes in memory. The flags are affected based on the result.

'ADI F3H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(A)	45H	38H
(F)	any values	Cy = 1, AC= 0, S = 0, P = 0, Z = 0

Summary: ADI d8 (2 bytes; ADI F3H; 1 opcode)

7.1.4 Instruction Type INR R

INR is a mnemonic that stands for '**INc**R**e**ment' and 'R' stands for any of the following registers or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to add 1 to the contents of R. The result of increment will be stored in R. All flags, except Cy flag, are affected depending on the result. Many times a register content is used as a counter. If Cy flag were to be affected during increment of a counter, it causes problems in many cases, as will be seen in program examples later. So by design, Cy flag is not affected by execution of this instruction.

As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'INR M' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(HL)	F850H	
(F850)	45H	46H
(F)	any values	Cy = no change, Ac = 0, S = 0, P = 0, Z = 0

If contents of F850H were FFH, it becomes 00H after execution of INR M.

Summary: INR R (1 byte; INR M; 8 opcodes)

7.1.5 Instruction Type ADC R

There are times when a user is required to add two numbers each of which is several bytes in size. For example, let us say it is needed to add the following two 16-bit numbers.

```
  1
 34 56H
+A2 F2H
-------
 D7 48H
```

In this example, the addition of 56H and F2H results in a sum of 48H with a carry of 1. Next, we have to add 34H and A2H along with this carry value of 1. To facilitate such an operation, 8085 provides instructions to add two numbers along with carry value.

ADC is a mnemonic that stands for 'ADd with Carry' and 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to add contents of R and Accumulator along with the carry value. The result of the addition will be stored in the Accumulator. As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'ADC E' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(E)	45H	
(A)	33H	79H
(F)	Cy = 1	Cy = 0, AC = 0, Z = 0, P = 0, S = 0
	others = any values	

Summary: ADC R (1 byte; ADC E; 8 opcodes)

7.1.6 Instruction Type ACI d8

ACI is a mnemonic that stands for '**A**dd with **C**arry **I**mmediate to accumulator' and 'd8' stands for any 8-bit data. This instruction is used to add 8-bit immediate data to the accumulator along with the carry value. The result of the addition will be stored in the accumulator. The flags are affected based on the result. It occupies 2 bytes in memory.

'ACI F3H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(A)	45H	38H
(F)	Cy = 0	Cy = 1, AC = 0, S = 0, P = 0, Z = 0
	others = any values	

Summary: ACI d8 (2 bytes; ACI F3H; 1 opcode)

■ 7.2 INSTRUCTIONS TO PERFORM SUBTRACTION

In this section we discuss the instructions to perform subtraction. In the subtraction of two numbers, 8085 imposes the restriction that Accumulator will have one operand from which the other operand specified by one of the following will be subtracted.

—Contents of an 8-bit register;
—Contents of memory location pointed by HL pair;
—Eight-bit immediate data.

7.2.1 Instruction Type SUB R

SUB is a mnemonic that stands for '**SUB**tract contents of R from Accumulator'. 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to subtract contents of R from Accumulator. The result of the subtraction will be stored in the Accumulator. As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'SUB E' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with examples.
Example 1:

	Before	***After***
(A)	F3H	AEH
(E)	45H	
(F)	any values	Cy = 0, AC = 0, S = 1, P = 0, Z = 0

Internally, 8085 performs this subtraction by adding the 2's complement of 45H to F3H, as shown below.

```
Cy Ac
 1 0
   F 3H
+ B BH ← 2's carry element of 45H
-------
   A EH
```

The carry generated in this addition is complemented and stored as the Cy flag value. Thus the Cy flag value after the subtraction will be 0. Note that even though it is a subtract operation, Intel prefers to use the terms carry and auxiliary carry in place of borrow and auxiliary borrow.

Example 2

	Before	***After***
(A)	30H	10H
(E)	20H	
(F)	any values	Cy = 0, AC= 1, S = 1, P = 0, Z = 0

Internally, 8085 performs this subtraction by adding the 2's complement of 20H to 30H, as shown in the following.

```
Cy Ac
 1 1
   3 0H
+ E 0H
-------
   1 0H
```

The carry generated in this addition is complemented and stored as the Cy flag value. Thus the Cy flag value after the subtraction will be 0.

Summary: SUB R (1 byte; SUB E; 8 opcodes)

7.2.2 Instruction Type SUI d8

SUI is a mnemonic that stands for '**SU**btract **I**mmediate from Accumulator' and 'd8' stands for any 8-bit data. This instruction is used to subtract 8-bit immediate data from the Accumulator. The result of the subtraction will be stored in the Accumulator. The flags are affected based on the result. It occupies 2 bytes in memory. 'SUI F3H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(A)	45H	52H
(F)	any values	Cy = 1, AC = 1, S = 0, P = 0, Z = 0

```
Cy Ac
 0 1
   4 5 H
+ 0 DH
-------
   5 2H
```

Summary: SUI d8 (2 bytes; SUI F3H; 1 opcode)

7.2.3 Instruction Type DCR R

DCR is a mnemonic, which stands for '**DeCR**ement' and 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to subtract 1 from the contents of R. The result of the increment will be stored in R. All flags, except Cy flag, are affected depending on the result. Many times a register content is used as a down counter. If Cy flag were to be affected during decrement of a counter, it causes problems in many cases, as will be seen in program examples later. So by design, Cy flag is not affected by the execution of this instruction.

As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'DCR M' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown in the following with an example.

	Before	***After***
(HL)	F850H	
(F850)	45H	44H
(F)	any values	Cy = no change, AC= 1, S = 0, P = 1, Z = 0

Internally, 8085 performs this decrement operation by adding the 2's complement of 01H to 45H. If contents of F850H were 00H, it becomes FFH after execution of DCR M.

Summary: DCR R (1 byte; DCR M; 8 opcodes)

7.2.4 Instruction Type SBB R

There are times when a user is required to subtract two numbers each of which is several bytes in size. For example, let us say it is needed to perform the following subtraction.

```
  -1
  34 56H
- 12 F2H
--------
  21 64H
```

In this example, the subtraction of 56H and F2H results in 64H with a borrow of 1. Next, we have to subtract 34H and 12H along with this borrow value of 1. To facilitate such an operation, 8085 provides instructions to subtract two numbers along with the borrow value.

SBB is a mnemonic that stands for '**SuB**tract with **B**orrow' and 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to subtract contents of R from accumulator, along with the carry value. The result of the subtraction will be stored in the Accumulator. As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'SBB E' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(E)	45H	
(A)	33H	EDH
(F)	Cy = 1	Cy = 1, AC= 0, Z = 0, P = 1, S = 1
	others = any values	

Summary: SBB R (1 byte; SBB E; 8 opcodes)

7.2.5 Instruction Type SBI d8

SBI is a mnemonic that stands for '**S**ubtract with **B**orrow **I**mmediate from Accumulator' and 'd8' stands for any 8-bit data. This instruction is used to subtract 8-bit immediate data from the Accumulator along with the carry value. The result of subtraction will be stored in the Accumulator. The flags are affected based on the result. It occupies 2 bytes in memory.

'SBI F3H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(A)	45H	52H
(F)	Cy = 0	Cy = 1, AC = 1, S = 0, P = 0, Z = 0
	others = any values	

Summary: SBI d8 (2 bytes; SBI F3H; 1 opcode)

■ 7.3 INSTRUCTION TYPE INX rp

INX is a mnemonic that stands for '**IN**crement e**X**tended register' and 'rp' stands for any of the following register pairs.

rp = BC, DE, or HL

This instruction is used to add 1 to the contents of rp. The result of the increment will be stored in rp. Note that flags are not at all affected by the execution of this instruction. A register pair is generally used to store a memory address. If flags were to be affected during increment of a memory address, it causes problems in many cases, as will be seen in program examples later. So by design, flags are not affected by execution of this instruction.

As rp can have any of the three values, there are three opcodes for this type of instruction. It occupies only 1 byte in memory. 'INX B' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(BC)	F850H	F851H

It may appear that INX B is functionally same as INR C in this example. But, if content of BC was F8FFH, it becomes F900H after execution of INX B. If content of BC was F8FFH, it becomes F800H after execution of INR C. Thus, basically, INX instruction increments a 16-bit quantity, whereas INR increments a 8-bit quantity.

Summary: INX rp (1 byte; INX B; 3 opcodes)

■ 7.4 INSTRUCTION TYPE DCX rp

DCX is a mnemonic, which stands for '**D**e**C**rement e**X**tended register' and 'rp' stands for any of the following register pairs.

rp = BC, DE, or HL

This instruction is used to subtract 1 from the contents of rp. The result of the decrement will be stored in rp. Note that flags are not at all affected by the execution of this instruction. A register pair is generally used to store a memory address. If flags were to be affected during decrement of a memory address, it causes problems in many cases, as will be seen in program examples later. So by design, flags are not affected by execution of this instruction.

As rp can have any of the three values, there are three opcodes for this type of instruction. It occupies only 1 byte in memory. 'DCX B' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(BC)	F850H	F84FH

It may appear that DCX B is functionally same as DCR C in this example. But, if content of BC was F900H, it becomes F8FFH after execution of DCX B. If content of BC was F900H, it becomes F9FFH after execution of DCR C. Thus, basically, DCX instruction decrements a 16-bit quantity, whereas DCR decrements a 8-bit quantity.

Summary: DCX rp (1 byte; DCX B; 3 opcodes)

■ 7.5 INSTRUCTION TYPE DAD rp

Intel 8085 is basically an 8-bit microprocessor. But the designers have provided instructions to perform 16-bit additions also. As the internal architecture is only 8 bits, this instruction easily takes double the time needed to add two 8-bit numbers.

DAD is a mnemonic, which stands for '**D**ouble **AD**d' and 'rp' stands for any of the following register pairs.

rp = BC, DE, or HL

This instruction is used to add contents of rp to HL. The result of the addition will be stored in HL. Thus in this instruction, HL is used as a 16-bit accumulator. Only Cy flag is affected depending on the result.

If other flags were to be affected during execution of this instruction, it causes problems in many cases, as will be seen in program examples later. So by design, flags other than Cy, are not affected by execution of this instruction.

As rp can have any of the three values, there are three opcodes for this type of instruction. It occupies only 1 byte in memory. 'DAD B' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(HL)	1234H	0A84H
(BC)	F850H	
(F)	any values	Cy = 1, no change in other flags

Summary: DAD rp (1 byte; DAD B; 3 opcodes)

■ 7.6 DECIMAL ADDITION IN 8085

In a digital computer everything is represented using 0s and 1s only. For example, an instruction will have a code using only 0s and 1s. Data is also represented using 0s and 1s. Data can be of different types like unsigned numbers, signed numbers, floating point numbers, binary coded decimal (BCD) numbers, etc. Thus, a series of 0s and 1s will acquire a value based on the interpretation.

7.6.1 BCD Numbers

Many a time, we are required to represent decimal numbers in a computer, and perform arithmetic on these numbers. For example, we may be required to total the marks a student has obtained in five different subjects, where obviously, the marks are awarded in decimal notation.

For this purpose BCD code is extensively used. In BCD notation, 4 bits are used to code a digit, and so two digits of information is stored in a byte. For example, decimal 39 is represented in BCD as 0011 1001. Codes 1010, 1011, 1100, 1101, 1110, and 1111 are illegal in BCD notation. Similarly decimal 1,024 is represented in BCD as 0001 0000 0010 0100. Same decimal 1,024 is represented in binary as 0000 0100 0000 0000. As stated earlier, the value of a series of 0s and 1s depends on the interpretation.

7.6.2 DAA Instruction

Suppose we want to add the two decimal numbers 38 and 45. They will be represented in BCD as 0011 1000 and 0100 0101. The addition results in 0111 1101. If we interpret this result as a BCD number, the answer is incorrect as well as illegal. This is because, 1101 is an illegal BCD number. This is where the DAA instruction proves its usefulness. All that is required to be done is to add the BCD numbers and store the result in A, and then execute the DAA instruction.

The working of DAA instruction depends on the contents of the AL register, Cy, and AC flags. In effect, it adds 00H, 06H, 60H, or 66H to accumulator so as to get the correct BCD answer in the Accumulator.

If the LS hex digit in A is <= 9 and AC flag is 0, the LS hex digit value will not be altered.
If the LS hex digit is > 9, or if AC flag is set to 1, it adds 6 to the LS hex digit of A. It increments the MS hex digit if this addition resulted in a carry to the MS digit position. In this process, the Cy flag will be set to 1 if the MS hex digit was incremented from F to 0.
Now, if the MS hex digit is <= 9 and Cy flag is 0, the MS hex digit will not be altered, and Cy flag is reset to 0.
If the MS hex digit is > 9, or if Cy flag is set to 1, it adds 6 to the MS hex digit of A and sets Cy flag to 1.

Note that DAA instruction cannot be used for decimal subtraction. Intel 8085 does not provide an instruction for decimal subtraction. So a series of instructions are to be executed to perform decimal subtraction.

A number of examples are provided here to explain decimal addition using DAA instruction.

Example 1: Let us say, we add 45 BCD and 38 BCD, and store the result 7DH in A. In this case, Cy flag = 0 and AC flag = 0. But as D is an invalid BCD code, the DAA instruction adds 06H to A. Thus, we get 83H in A, which will now be interpreted as 83 BCD, the correct sum of 45 and 38.

Example 2: Let us say, we add 63 BCD and 88 BCD, and store the result EBH in A. In this case, Cy flag = 0 and AC flag = 0. But as both E and B are invalid BCD codes, the DAA instruction adds 66H to A. Thus, we get 51H in A, and Cy flag will be set to 1. This will now be interpreted as 51 BCD with Cy flag = 1, the correct sum of 63 and 88.

Example 3: Let us say, we add 53 BCD and 36 BCD, and store the result 89H in A. In this case, Cy flag = 0 and AC flag = 0. Since both 8 and 9 are valid BCD codes, the DAA instruction does not add anything to A. In other words, it is same as adding 00 to AL. The result in A will now be interpreted as 89 BCD, the correct sum of 53 and 36.

Example 4: Let us say, we add 99 BCD and 88 BCD, and store the result 21H in A. In this case, Cy flag = 1 and AC flag = 1. So, although both 2 and 1 are valid BCD codes, the DAA instruction adds 66H to A. Thus, we get 87H in A, and Cy flag will be set to 1. The result in A will now be interpreted as 87 BCD with Cy flag = 1, the correct sum of 99 and 88.

Example 5: Let us say, we add 63 BCD and 42 BCD, and store the result A5H in A. In this case, Cy flag = 0 and AC flag = 0. But as A is an invalid BCD code, the DAA instruction adds 60H to AL. Thus we get 05H in A, with Cy flag = 1, which will now be interpreted as 05 BCD with Cy flag = 1, the correct sum of 63 and 42.

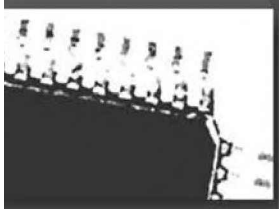
1. Describe the flags available in 8085.
2. Assume that before the execution of any instruction we have (A) = 65H, (B) = B2H, (H) = F9H, (L) = 50H, Cy flag = 1, and content of memory location F950H is 38H. What is the value of A register and the value of different flags after the execution of each of the following?

   a. ADD L
   b. ADC B
   c. SUI 56H
   d. SBB M and
   e. SUB H

3. Distinguish between the following pairs of instructions.

   a. ADD H and ADC H
   b. ADI 53H and ACI 53H
   c. ADC M and ADC H
   d. ADD M and DAD H
   e. SUB M and SUB L
   f. INX B and INR C

4. Write a 8085 assembly language program to perform addition of BC and HL, and store in HL without using DAD instruction.
5. Explain the working of DAA instruction with examples.
6. Write a 8085 assembly language program to perform decimal addition without using DAA instruction.
7. Write a 8085 assembly language program to perform decimal subtraction.



8 Logical Group of Instructions

- Instructions to perform AND operation
  - *Instruction type ANA R*
  - *Instruction type ANI d8*
- Instructions to perform OR operation
  - *Instruction type ORA R*
  - *Instruction type ORI d8*
- Instructions to perform Exclusive OR operation
  - *Instruction type XRA R*
  - *Instruction type XRI d8*
- Instruction to complement accumulator
- Instructions to complement/set Cy flag
  - *Instruction type CMC*
  - *Instruction type STC*
- Instructions to perform compare operation
  - *Instruction type CMP R*
  - *Instruction type CPI d8*
- Instructions to rotate accumulator
  - *Instruction type RLC*
  - *Instruction type RAL*
  - *Instruction type RRC*
  - *Instruction type RAR*
- Questions

A total of 15 instruction types covering 43 instructions will be explained in this chapter. These include the various instruction types to perform AND, OR, exclusive OR, complement Accumulator, complement = set Cy flag, compare and rotate Accumulator operations.

■ 8.1 INSTRUCTIONS TO PERFORM 'AND' OPERATION

In the logical group of instructions, 8085 has instructions to perform AND, OR, Ex-OR, and NOT operations. It does not have instructions to perform NAND or NOR operations. This is because in controlling of peripherals, it is generally not required to perform NAND or NOR operations. In this section we discuss the instructions to perform AND operation.

In operations like AND, which need two operands, 8085 imposes the restriction that one of the operands must be in the Accumulator. The other operand can be one of the following.

- Contents of an 8-bit register;
- Contents of memory location pointed by HL pair;
- Eight-bit immediate data.

The AND operation performs bit-wise AND of the two operands. If X is a bit of Accumulator, and Y is a bit of the other operand in the same bit position, the AND operation is performed as per the following truth table.

Truth Table for AND Operation

X	*Y*	*X AND Y*
0	0	0
0	1	0
1	0	0
1	1	1

It can be noticed that X AND 1 = X, and X AND 0 = 0. Thus, AND operation is used for selectively resetting to 0 some bits of the Accumulator. Bits of the Accumulator that are ANDed with 0s are reset to 0, and bits of the Accumulator, which are ANDed with 1s are not changed. Thus, if it is desired to reset MS 4 bits of Accumulator, AND Accumulator contents with 0FH.

The AND instruction affects the flags as follows.

- S, P, and Z flags are updated based on the result;
- Cy flag is reset to 0;
- AC flag is set to 1.

8.1.1 Instruction Type ANA R

ANA is a mnemonic, which stands for '**AN**d **A**ccumulator' and 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to AND contents of R with Accumulator. The result of AND operation will be stored in the Accumulator. As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'ANA E' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(E)	45H	
(A)	33H	01H
(F)	any values	Cy = 0, AC = 1, Z = 0, P = 0, S = 0

(A) = 33H = 0 0 1 1 0 0 1 1
AND
(E) = 45H = 0 1 0 0 0 1 0 1
0 0 0 0 0 0 0 1 = 01H

Summary: ANA R (1 byte; ANA E; 8 opcodes)

8.1.2 Instruction Type ANI d8

ANI is a mnemonic, which stands for '**AN**d **I**mmediate with Accumulator' and 'd8' stands for any 8-bit data. This instruction is used to AND 8-bit immediate data with the Accumulator. The result of ANDing will be stored in the Accumulator. The S, P, and Z flags are affected based on the result. Cy is reset to 0, and AC is set to 1. It occupies 2 bytes in memory.

'ANI F3H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(A)	45H	41H
(F)	any values	Cy = 0, AC = 1, S = 0, P = 1, Z = 0

Summary: ANI d8 (2 bytes; ANI F3H; 1 opcode)

■ 8.2 INSTRUCTIONS TO PERFORM 'OR' OPERATION

The OR operation performs bit-wise OR of the two operands. If X is a bit of Accumulator, and Y is a bit of the other operand in the same bit position, the OR operation is performed as per the following truth table.

Truth Table for OR Operation

X	*Y*	*X OR Y*
0	0	0
0	1	1
1	0	1
1	1	1

It can be noticed that X OR 1 = 1, and X OR 0 = X. Thus, OR operation is used for selectively setting some bits of the Accumulator to 1. Bits of the Accumulator, which are ORed with 1s are set to 1, and bits of the Accumulator, which are ORed with 0s are not changed. Thus, if it is desired to set MS 4 bits of Accumulator, OR Accumulator contents with F0H.

The OR instruction affects the flags as follows.

– S, P, and Z flags are updated based on the result;
– Cy and AC flags are reset to 0.

8.2.1 Instruction Type ORA R

ORA is a mnemonic, which stands for '**OR** Accumulator' and 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to OR contents of R with the Accumulator. The result of OR operation will be stored in the Accumulator. As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'ORA E' is an example instruction of this type. It is an 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(E)	45H	
(A)	33H	77H
(F)	any values	Cy = 0, Ac = 0, Z = 0, P = 1, S = 0

(A) = 33H = 0 0 1 1 0 0 1 1
OR
(E) = 45H = 0 1 0 0 0 1 0 1
0 1 1 1 0 1 1 1 = 77H

Summary: ORA R (1 byte; ORA E; 8 opcodes)

8.2.2 Instruction Type ORI d8

ORI is a mnemonic that stands for '**OR** Immediate with Accumulator' and 'd8' stands for any 8-bit data. This instruction is used to OR 8-bit immediate data with the Accumulator. The result of ORing will be stored in the Accumulator. The S, P, and Z flags are affected based on the result. Cy and AC are reset to 0. It occupies 2 bytes in memory. 'ORI F3H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(A)	45H	F7H
(F)	any values	Cy = 0, AC = 0, S = 1, P = 0, Z = 0

Summary: ORI d8 (2 bytes; ORI F3H; 1 opcode)

■ 8.3 INSTRUCTIONS TO PERFORM 'EXCLUSIVE OR' OPERATION

The Ex-OR operation performs bit-wise Ex-OR of the two operands. If X is a bit of Accumulator, and Y is a bit of the other operand in the same bit position, the Ex-OR operation is performed as per the following truth table.

Truth Table for Ex-OR Operation

X	*Y*	*X Ex-OR Y*
0	0	0
0	1	1
1	0	1
1	1	0

It can be noticed that X Ex-OR 1 = X*, and X Ex-OR 0 = X. Thus, Ex-OR operation is used for selectively complementing some bits of the Accumulator. Bits of the Accumulator, which are Ex-ORed with 1s are complemented, and bits of the Accumulator, which are Ex-ORed with 0s are not changed. Thus, if it is desired to complement MS 4 bits of Accumulator, Ex-OR Accumulator contents with F0H.

The Ex-OR instruction affects the flags as follows.

– S, P, and Z flags are updated based on the result;
– Cy and AC flags are reset to 0.

Instruction Type XRA R

XRA is a mnemonic that stands for '**eX**clusive **OR** accumulator' and 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to Ex-OR contents of R with the Accumulator. The result of Ex-OR operation will be stored in the Accumulator. As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'XRA E' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(E)	45H	
(A)	33H	76H
(F)	any values	Cy = 0, AC = 0, Z = 0, P = 0, S = 0

(A) = 33H = 0 0 1 1 0 0 1 1
Ex-OR
(E) = 45H = 0 1 0 0 0 1 0 1

0 1 1 1 0 1 1 0 = 76H

Summary: XRA R (1 byte; XRA E; 8 opcodes)

8.3.2 Instruction Type XRI d8

XRI is a mnemonic that stands for '**eX**clusive **OR** **I**mmediate with Accumulator' and 'd8' stands for any 8-bit data. This instruction is used to Ex-OR 8-bit immediate data with the Accumulator. The result of Ex-ORing will be stored in the Accumulator. The S, P, and Z flags are affected based on the result. Cy and AC are reset to 0. It occupies 2 bytes in memory. 'XRI F3H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(A)	45H	B6H
(F)	any values	Cy = 0, AC = 0, S = 1, P = 0, Z = 0

Summary: XRI d8 (2 bytes; XRI F3H; 1 opcode)

■ 8.4 INSTRUCTION TO COMPLEMENT ACCUMULATOR

The complement instruction in 8085 has the mnemonic CMA. It stands for '**C**o**M**plement the **A**ccumulator'. It performs 1's complement operation on the contents of Accumulator, and the result is stored back in the Accumulator. Notice that only Accumulator contents can be complemented and not any other register. Flags are not affected by the execution of this instruction. It occupies only 1 byte in memory. The result of execution of this instruction is shown below with an example.

	Before	*After*
(A)	45H	BAH

Summary: CMA (1 byte; CMA; 1 opcode)

■ 8.5 INSTRUCTIONS TO COMPLEMENT/SET 'Cy' FLAG

Intel 8085 also provides instructions to complement the Cy flag, and set the Cy flag to the 1 state. But it does not have an instruction to reset the Cy flag to 0. If it is desired to reset Cy flag to 0, the method is to set it to 1 and then complement it. Notice that no other flag can be set or complemented.

8.5.1 Instruction Type CMC

CMC stands for '**C**o**M**plement the **C**arry flag'. It performs complement operation on the Cy flag, and the result is stored back in the Cy flag. The result of execution of this instruction is shown below with an example.

	Before	*After*
(Cy)	1	0

Summary: CMC (1 byte; CMC; 1 opcode)

8.5.2 Instruction Type STC

STC stands for '**S**e**T** the **C**arry flag'. It sets the Cy flag to the 1 state, immaterial of its earlier value. The result of execution of this instruction is shown below with examples.

Example 1

	Before	*After*
(Cy)	1	1

Example 2

	Before	*After*
(Cy)	0	1

Summary: STC (1 bytes; STC; 1 opcode)

■ 8.6 INSTRUCTIONS TO PERFORM COMPARE OPERATION

A compare instruction compares two operands, and affects the status flags' values depending on the result of the comparison. In this operation, 8085 imposes the restriction that one of the operands must be in the Accumulator. The other operand can be one of the following.

- Contents of an 8-bit register;
- Contents of memory location pointed by HL pair;
- Eight-bit immediate data.

The compare instruction actually computes the value of the Accumulator contents minus other operand. The original values of the operands are not changed. The result is stored in a register that is not accessible to the programmer. Based on the result, all the flags are affected.

It is similar to comparing the heights of two people person 1 and person 2. After the comparison, the heights remain unaltered. But, we would have come to one of the following conclusions.

- Both are of same height;
- Person_1 is taller;
- Person_1 is shorter.

8.6.1 Instruction Type CMP R

CMP is a mnemonic that stands for '**CoMP**are Accumulator' and 'R' stands for any of the following registers, or memory location M pointed by HL pair.

R = A, B, C, D, E, H, L, or M

This instruction is used to compare contents of the Accumulator with R. The result of compare operation will be stored in the Temp register. Temp is an internal register that is not accessible to the programmer. As R can have any of the eight values, there are eight opcodes for this type of instruction. It occupies only 1 byte in memory. 'CMP E' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with examples.

Example 1

	Before	*After*
(A)	F5H	
(E)	25H	
(Temp)	45H	D0H
(F)	any values	Cy= 0, AC= 1, Z= 0, P= 0, S= 1

Example 2

	Before	***After***
(A)	50H	
(E)	70H	
(Temp)	45H	E0H
(F)	any values	Cy = 1, AC = 1, Z = 0, P = 0, S = 1

Example 3

	Before	***After***
(A)	50H	
(E)	50H	
(Temp)	45H	00H
(F)	any values	Cy = 0, AC = 1, Z = 1, P = 1, S = 0

Example 4

	Before	***After***
(A)	F5H	
(E)	D5H	
(Temp)	45H	20H
(F)	any values	Cy = 0, AC = 1, Z = 0, P = 0, S = 0

Example 5

	Before	***After***
(A)	25H	
(E)	F5H	
(Temp)	45H	30H
(F)	any values	Cy = 1, AC = 1, Z = 0, P = 1, S = 0

Summary: CMP R (1 byte; CMP E; 8 opcodes)

8.6.2 Instruction Type CPI d8

CPI is a mnemonic that stands for '**C**om**P**are **I**mmediate with Accumulator' and 'd8' stands for any 8-bit data. This instruction is used to compare Accumulator with 8 bit immediate data. The result of the comparison will be stored in an internal register not accessible to the programmer. All the flags are affected based on the result. It occupies 2 bytes in memory. 'CPI F5H' is an example instruction of this type. It is a 2-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	***After***
(A)	25H	
(Temp)	45H	30H
(F)	any values	Cy = 1, AC = 1, Z = 0, P = 1, S = 0

Summary: CPI d8 (2 bytes; CPI F5H; 1 opcode)

■ 8.7 INSTRUCTIONS TO ROTATE ACCUMULATOR

Intel 8085 provides instructions to rotate Accumulator contents left or right. It is to be noted here that rotate operation can be performed only on Accumulator contents. These instructions are explained as follows.

8.7.1 INSTRUCTION TYPE RLC

RLC stands for '**R**otate **L**eft Accumulator'. It rotates the Accumulator contents to the left by 1-bit position. Fig. 8.1 illustrates this operation.

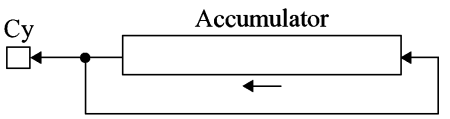


Fig. 8.1
Rotate left without involving Cy in rotation

As can be seen, after rotate left operation, the bit that moves out from the MS bit position goes to the vacancy created in the LS bit position. Also, Cy flag gets a copy of the bit moved out from the MS bit position. Notice that Cy flag is not involved in the rotation, and it is only 8-bit rotation of accumulator contents. Only Cy flag is affected by this instruction execution.

The instruction is useful in the following ways.

- To check the value of the MS bit of Accumulator, perform rotate left, and note the cy flag value.
- To perform multiplication by 2, rotate the Accumulator to left. It works correctly for unsigned numbers, as long as the MS bit of Accumulator is a 0 before rotation. For multiplication by 2^n, perform rotate left n times.

The result of execution of this instruction is shown below with examples.

Example 1

	Before	***After***
(A)	24H	48H
(Cy)	1	0

Note that accumulator value is doubled.

Example 2

	Before	***After***
(A)	84H	09H
(Cy)	0	1

Note that accumulator value is not doubled in this case because MS bit of accumulator was a 1 before rotation.

8.7.2 Instruction Type RAL

RAL stands for '**R**otate **A**ccumulator **L**eft involving Cy flag in rotation'. It rotates the Accumulator contents to the left by 1-bit position. Fig. 8.2 illustrates this operation.

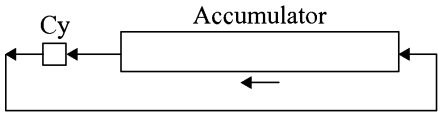


Fig. 8.2
Rotate left involving carry in rotation

As can be seen, after rotate left operation, the bit moves out from the MS bit position and goes to the Cy flag, and in the process moves out the earlier carry bit to the vacancy created in the LS bit position. Notice that Cy flag is involved in the rotation, and it is 9-bit rotation of Accumulator and Cy contents. Only Cy flag is affected by this instruction execution.

The instruction is useful in the following ways.

- To check the value of the MS bit of Accumulator, perform rotate left, and note the Cy flag value.
- To perform multiplication by 2, rotate the Accumulator to the left. It works correctly for unsigned numbers, as long as the MS bit of Accumulator and Cy flag are 0 before rotation. For multiplication by 2^n, perform rotate left n times.
- To introduce a new bit value to the LS bit position, put this bit value in the Cy flag and then execute this instruction.

The result of execution of this instruction is shown below with examples.

Example 1

	Before	***After***
(A)	24H	48H
(Cy)	0	0

Note that Accumulator value is doubled.

Example 2

	Before	***After***
(A)	84H	08H
(Cy)	0	1

Note that Accumulator value is not doubled in this case because MS bit of Accumulator was a 1 before rotation.

Example 3

	Before	***After***
(A)	24H	49H
(Cy)	1	0

Note that Accumulator value is not doubled in this case because Cy flag was a 1 before rotation.

8.7.3 Instruction Type RRC

RRC stands for '**R**otate **R**ight Accumulator'. It rotates the Accumulator contents to the right by 1-bit position. Fig. 8.3 illustrates this operation.

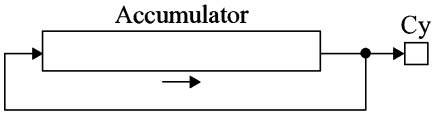


Figure. 8.3
Rotate right without involving carry in rotation

As can be seen, after rotate right operation, the bit moves out from the LS bit position and goes to the vacancy created in the MS bit position. Also, Cy flag gets a copy of the bit moved out from the LS bit position. Notice that the Cy flag is not involved in the rotation, and it is only 8-bit rotation of Accumulator contents. Only Cy flag is affected by this instruction execution.

The instruction is useful in the following ways.

- To check the value of the LS bit of Accumulator, perform rotate right, and note the Cy flag value.
- To perform division by 2, rotate the Accumulator to the right. It works correctly for unsigned numbers, as long as the LS bit of Accumulator is a 0 before rotation. For division by 2^n, perform rotate right n times.

The result of execution of this instruction is shown below with examples.

Example 1

	Before	***After***
(A)	48H	24H
(Cy)	1	0

Note that Accumulator value is halved.

Example 2

	Before	***After***
(A)	49H	A4H
(Cy)	0	1

Note that Accumulator value is not halved in this case because LS bit of accumulator was a 1 before rotation.

8.7.4 Instruction Type RAR

RAR stands for '**R**otate **A**ccumulator **R**ight involving Cy flag in rotation'. It rotates the Accumulator contents to the right by 1-bit position. Fig. 8.4 illustrates this operation.

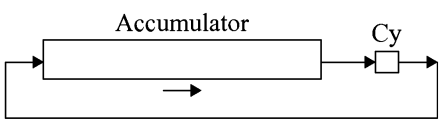


Figure. 8.4
Rotate right involving carry in rotation

As can be seen, after rotate right operation, the bit moves out from the LS bit position and goes to the Cy flag, and in the process moves out the earlier carry bit to the vacancy created in the MS bit position. Notice that Cy flag is involved in the rotation, and it is 9-bit rotation of Accumulator and Cy contents. Only Cy flag is affected by this instruction execution.

The instruction is useful in the following ways.

- To check the value of the LS bit of Accumulator, perform rotate right, and note the Cy flag value.
- To perform division by 2, rotate the Accumulator to the right. It works correctly for unsigned numbers, as long as the LS bit of Accumulator and Cy flag are 0 before rotation. For division by 2^n, perform rotate right n times.
- To introduce a new bit value to the MS bit position, put this bit value in the Cy flag and then execute this instruction.

The result of execution of this instruction is shown below with examples.

Example 1

	Before	***After***
(A)	48H	24H
(Cy)	0	0

Note that Accumulator value is halved.

Example 2

	Before	***After***
(A)	49H	24H
(Cy)	0	1

Note that Accumulator value is not halved in this case because LS bit of Accumulator was a 1 before rotation.

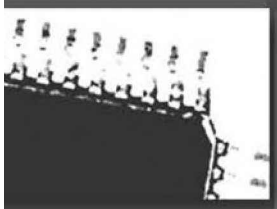
Example 3

	Before	***After***
(A)	48H	A4H
(Cy)	1	0

Note that Accumulator value is not halved in this case because Cy flag was a 1 before rotation.

1. Write an 8085 assembly language program, which takes the data from memory location X, and sets to 1 all the odd numbered bits of this byte, and stores the result at memory location Y.
2. Write an 8085 assembly language program, which takes the data from memory location X, and resets to 0 all the even numbered bits of this byte, and stores the result at memory location Y.
3. Write an 8085 assembly language program that takes the data from memory location X, and complements the 4 bits in the middle of this byte, and stores the result at memory location Y.
4. Write an 8085 assembly language program, which takes the data from memory location X, and sets to 1 the MS 2 bits, complements the 4 bits in the middle of this byte, and resets to 0 the LS 2 bits, and stores the result at memory location Y.
5. Write an 8085 assembly language program, which takes the data from memory location X, and multiplies this byte by 4, and stores the result at memory location Y.

6. Write an 8085 assembly language program, which takes the data from memory location X, and divides this byte by 8, and stores the result at memory location Y.
7. Write an 8085 assembly language program, which takes the data from memory location X, and multiplies this byte by 10, and stores the result at memory location Y.
8. Assume that before the execution of any instruction we have (A) = 65H, (B) = B2H, (H) = F9H, (L) = 50H, Cy flag = 1, and content of memory location F950H is 38H. What is the value of A register and value of different flags after the execution of each of the following.
   a. CMP B
   b. CMP M
   c. CPI 55H
   d. RAL
   e. RRC
   f. CMA
   g. CMC
9. Distinguish between the following pairs of instructions.
   a. RAL and RLC
   b. RRC and RLC
   c. CMA and CMC
   d. XRA M and ORA M
10. Explain the working of CMP instruction with examples.



9 NOP and Stack Group of Instructions

- Stack and the stack pointer
  - *Reading from the stack*
  - *Writing to the stack*
- Instruction type POP rp
- Instruction type PUSH rp
- Instruction type LXI SP, d16
- Instruction type SPHL
- Instruction type XTHL
- Instruction type INX SP
- Instruction type DCX SP
- Instruction type DAD SP
- Instruction type NOP
- Questions

A total of nine instruction types covering 15 instructions will be explained in this chapter.

The NOP and stack group of instructions are discussed in depth in this chapter. In the first half, the role of the stack pointer (SP) is explained and the various instruction types are dealt with in the remaining part of the chapter.

9.1 STACK AND THE STACK POINTER

Stack is a LIFO (last in, first out) data structure implemented in the RAM area and is used to store addresses and data, when the microprocessor branches to a subroutine. Subroutines are discussed in the next chapter.

In the programmer's view of 8085, only the general purpose registers A, B, C, D, E, H, and L, and the Flags registers were discussed so far. But in the complete programmer's view of 8085, there are two more special purpose registers, each of 16-bit width. They are the stack pointer, SP, and the program counter, PC. The role of PC will be explained in the next chapter. The complete programmer's view of 8085 is shown in Fig. 9.1.

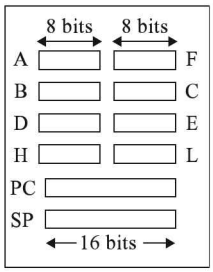


Fig. 9.1
Programmer's view of 8085

SP is a special purpose 16-bit register. It contains a memory address. Suppose SP contents are FC78H, then the 8085 interprets it as follows.

Memory locations FC78H, FC79H, ... , FFFFH are having useful information. In other words, these locations are treated as filled locations. Memory locations FC77H, FC76H, ... , 0000H are not having any useful information. In other words, these locations are treated as empty locations.

Thus, the contents of SP specify the top most useful location in the stack. In other words, it indicates the memory location with the smallest address having useful information. This is pictorially represented in Fig. 9.2.

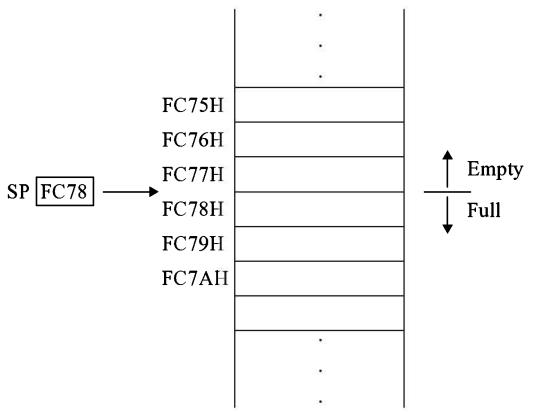


Fig. 9.2
Interpretation of SP contents

9.1.1 Reading from the Stack

Let us say SP contents are FC78H, and we want to read information from a stack location. In this case, we are not interested in reading from a location whose address is less than the memory address present in SP. This is because 8085 interprets them as useless information. For example, there is no point in reading useless information from memory location FC75H.

Right now, memory locations FC78H, FC79H, ... , FFFFH are all interpreted by 8085 to have useful information. Then, which one of these useful information to read? The 8085 reads from the top of the stack of useful information. The 8085 is designed to read always 2 bytes useful information from the top of stack. As such, in this case it reads from locations FC78H and FC79H. In 8085, this information, which is read can only be loaded into a register pair. This operation of loading a register pair by reading information from the stack top is called a POP operation.

Suppose we load register pair BC when SP contents was FC78H. Then, information from memory locations FC78H and FC79H are copied to BC pair. But after this pop operation, the contents of FC78H and FC79H are treated as useless by the 8085. This is because, a copy is there in the register pair BC

anyway! Thus only FC7AH, FC7BH, ... , FFFFH are useful locations. To indicate this, the SP contents are changed to FC7AH. This is done automatically by 8085 by incrementing SP contents by 2.

9.1.2 Writing to the Stack

Let us say SP contents are FC7AH, and we want to write information to a stack location. In this case, we are not interested in writing to a location whose address is equal or greater than the memory address present in SP. This is because the 8085 interprets them as having useful information, which should not be destroyed! For example, there is no point in overwriting and destroying useful information at memory location FD7AH. We should be writing into a location where there is presently useless information, and make it useful!

Right now, memory locations FC79H, FC78H, ... , 0000H are all interpreted by 8085 to have useless information. Then, which one of these useless locations to overwrite? The 8085 writes above the top of the stack of useful information. The 8085 is designed to write always into two locations of useless information just above the top of stack. As such, in this case it writes to locations FC79H and FC78H. In 8085, this information, which is written can only be coming from a register pair. This operation of storing a register pair by writing information above the stack top is called a PUSH operation.

Suppose we store register pair BC when SP contents were FC7AH, then, information from BC pair is stored in memory locations FC79H and FC78H. But after this push operation, the contents of FC79H and FC78H are treated as useful by the 8085. This is because nobody stores useless information! Thus FC78H, FC79H, ... , FFFFH are all useful locations. To indicate this, the SP contents are changed to FC78H. This is done automatically by 8085 by decrementing SP contents by 2.

■ 9.2 INSTRUCTION TYPE POP rp

This instruction loads register pair rp by popping out 2 bytes from the top of the stack. In previous chapters, 'rp' stood for any of the register pairs BC, DE, or HL. But as per the programmer's view in Fig. 9.1, we can treat combination of Accumulator and flags as one more register pair. This pair is generally called PSW, which stands for 'processor status word'. In the PSW, Accumulator is the MS byte, and Flags register is the LS byte.

Two bits are used in an opcode to specify a register pair. Thus actually four register pairs can be specified using 2-bit code for 'rp'. In fact, even SP can be treated as a register pair. Thus in all there are five register pairs. One may think that to specify one of them, 3 bits are needed. But 8085 opcodes use only 2 bits to specify a register pair as shown below. The 'rp' code of 11 specifies either SP or PSW, but not both. For example, in POP rp instruction, rp can be BC, DE, HL, or PSW. There is no instruction like POP SP. Similarly, in LXI rp instruction, rp can be BC, DE, HL, or SP. There is no instruction like LXI PSW.

rp code	**register pair**
0 0	BC
0 1	DE
1 0	HL
1 1	SP or PSW, but never both.

Thus, in POP rp instruction 'rp' stands for one of the following register pairs.

rp = BC, DE, HL, or PSW

As rp can have any of the four values, there are four opcodes for this type of instruction. It occupies only 1 byte in memory. 'POP PSW' is an example instruction of this type. It is an 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(SP)	FE06H	FE08H
(FE06)	66H	
(FE07)	55H	
(A)	33H	55H
(F)	44H	66H

Note that POP PSW instruction is useful in loading the flags register with any desired value. Suppose it is desired to load flags register with the value 66H, the following instructions have to be executed.

```
MVI C, 66H
PUSH B
POP PSW
```

Summary: POP rp (1 byte; POP PSW; 4 opcodes)

■ 9.3 INSTRUCTION TYPE PUSH rp

This instruction stores contents of register pair rp by pushing it into two locations above the top of the stack. **rp** stands for one of the following register pairs.

rp = BC, DE, HL, or PSW

As rp can have any of the four values, there are four opcodes for this type of instruction. It occupies only 1 byte in memory. 'PUSH B' is an example instruction of this type. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.

	Before	*After*
(B)	77H	
(C)	88H	
(SP)	FB06H	FB04H
(FB05)	66H	77H
(FB04)	55H	88H

Summary: PUSH rp (1 byte; PUSH B; 4 opcodes)

Suppose we execute the following instructions, with SP contents as FB08H to start with.

```
PUSH B
PUSH D
PUSH H
PUSH PSW
```

Then the stack contents will be as shown overleaf after the execution of the previously mentioned instructions.

FB00H	(F)
FB01H	(A)
FB02H	(L)
FB03H	(H)
FB04H	(E)
FB05H	(D)
FB06H	(C)
FB07H	(B)

The SP contents are changed to FB00H. As we start pushing, the information starts getting stacked one above the other. This justifies the name 'stack' for this data structure in RAM. Notice that the stack has to be implemented in RAM, as we need to perform write operation on the stack, called the PUSH operation.

In the earlier example, PSW contents were pushed above the stack top in the end. Now if we perform pop operation, this information comes out first. Thus, a stack is a LIFO (Last In First Out) data structure. Of course, it can also be equivalently termed as first in last out (FILO).

In summary, SP is a 16-bit register inside 8085. Its contents specify the topmost filled memory location in the stack. Stack is a LIFO data structure implemented in RAM. In push or pop operations the data transfer is between a register pair and the stack.

The stack grows towards the address 0000H, as push operations are performed. The SP content is decremented by two for every push operation. It shrinks towards the address FFFFH, as pop operations are performed. The SP contents are incremented by two for every pop operation.

SP is generally loaded with the highest RAM address +1. For example, if we have RAM from C000H to C7FFH, then SP is loaded with C800H at the beginning of the user program. The program is generally loaded on a kit starting from the lowest RAM address. This allows the user program to expand towards higher addresses and the stack to grow towards lower addresses, without overlapping.

■ 9.4 INSTRUCTION TYPE LXI SP, d16

Let us say, SP contents is FC00H and it is desired to pop contents of FC06H and FC07H to BC register pair. One way of achieving this objective is by executing the following sequence of instructions.

```
POP B; pop contents of FC00 and FC01 to BC pair;
POP B; pop contents of FC02 and FC03 to BC pair;
POP B; pop contents of FC04 and FC05 to BC pair;
POP B; finally pop contents of FC06 and FC07 to BC pair.
```

However, this is a laborious process. Loading SP with FC06H, and then popping to BC pair can achieve the same very efficiently. When SP is loaded with FC06H, 8085 interprets that FC06H, FC07H, ... , FFFFH are all filled locations. Then POP B instruction execution results in popping contents of FC06H and FC07H to BC pair.

LXI SP, d16 instruction is a special case of LXI rp, d16 which was discussed in the chapter on data transfer group of instructions. Thus LXI SP instruction is used to load 16-bit immediate data to the SP. It occupies 3 bytes in memory. 'LXI SP FC06H' is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown in the following example.

	Before	*After*
(SP)	FC00H	FC06H

Summary: LXI SP d16 (3 bytes; LXI SP, FC06H; 1 opcode)

■ 9.5 INSTRUCTION TYPE SPHL

This instruction loads the stack pointer with the contents of register pair HL. It is an indirect way of loading the stack pointer, and as such, it is not quite commonly used. It occupies only 1 byte in memory, compared to LXI SP instruction, which is 3 bytes long. Because of this advantage, SPHL can be useful when SP is required to be initialized to a specific value a number of times in a program.

The result of execution of this instruction is shown below with an example.

	Before	*After*
(HL)	FB50H	
(SP)	F900H	FB50H

Summary: SPHL (1 byte; SPHL; 1 opcode)

■ 9.6 INSTRUCTION TYPE XTHL

XTHL is a mnemonic that stands for 'e**X**change **T**op of stack with **HL**'. This instruction exchanges the contents of the top two locations of the stack with the contents of register pair HL. Note that it is not exchange of SP with HL.

It occupies only 1 byte in memory. The result of execution of this instruction is shown below with an example. Note that SP contents remain unchanged. It is neither decremented nor incremented.

	Before	*After*
(SP)	F900H	
(HL)	FB50H	3525H
(F900)	25H	50H
(F901)	35H	FBH

Summary: XTHL (1 byte; XTHL; 1 opcode)

■ 9.7 INSTRUCTION TYPE INX SP

INX SP instruction is a special case of INX rp that was discussed in the chapter on arithmetic group of instructions. Thus INX SP instruction is used to increment the SP contents by 1. It occupies only 1 byte in memory. The result of execution of this is shown in the following example.

	Before	*After*
(SP)	FC00H	FC01H

Summary: INX SP (1 byte; INX SP; 1 opcode)

■ 9.8 INSTRUCTION TYPE DCX SP

DCX SP instruction is a special case of DCX rp that was discussed in the chapter on arithmetic group of instructions. Thus DCX SP instruction is used to decrement the SP contents by 1. It occupies only 1 byte in memory. The result of execution of this instruction is shown below with an example.

	Before	*After*
(SP)	FC00H	FBFFH

Summary: DCX SP (1 byte; DCX SP; 1 opcode)

■ 9.9 INSTRUCTION TYPE DAD SP

DAD SP instruction is a special case of DAD rp, which was discussed in the chapter on arithmetic group of instructions. Thus DAD SP instruction is used to add SP contents to HL contents, and store the result in HL. It occupies only 1 byte in memory. The result of execution of this instruction is shown below with an example. Only carry flag is affected depending on the result.

	Before	*After*
(HL)	0234H	FA84H
(SP)	F850H	
(F)	any values	Cy = 0, no change in other flags

Summary: DAD SP (1 byte; DAD SP; 1 opcode)

■ 9.10 INSTRUCTION TYPE NOP

NOP is an instruction that cannot be classified under any group discussed so far. For convenience, we discuss this instruction in this chapter.

NOP is a mnemonic that stands for '**N**o **OP**eration'. This instruction does not do anything! Still, it is quite useful. It occupies only 1 byte in memory. It has the opcode 00H. It is useful in the following cases.

1. NOP instruction is very useful for generating small-time delays of the order of a few microseconds. Let us say, the 8085 has to send information to two peripherals with a gap of approximately 4 μs. In such a case, after sending information to the first peripheral, it has to wait for 4 μs. This

is where an NOP instruction is very useful. Every instruction needs a certain time for instruction fetch and execution. These details are provided in a later chapter on 8085 architecture. NOP instruction uses up about 1.3 μs for the instruction fetch and execution. If there are three NOP instructions, then about 4 μs delay is generated. After these NOP instructions, information can be sent to the second peripheral.

2. NOP instruction is very useful when we are required to delete a few instructions in our program. Let us say we have a long program as shown in the following.

Memory address	**Contents**
F800H	LDA FE00H
F803H	MOV C, A
F804H	MOV D, A
F805H	MOV E, A
.	
.	
.	
FD56H	HLT

When the program is executed, let us say we do not get the desired result, because the program is wrong. Then we go about debugging the program in single step mode. Let us say, we come to the conclusion that the instruction 'MOV D, A' at location F804H should be removed. If we remove the instruction 'MOV D, A', then the program portion from memory location F805H to FD56H should be moved up by one position. This amounts to almost rewriting the whole program, as the error has occurred in the beginning of the program. Anyway, we do not have to lose heart, as NOP instruction is there to our rescue. We just have to replace 'MOV D, A' with NOP instruction! Then there is no need for moving up by one position the program portion from F805H to FD56H.

3. NOP instruction is very useful when we are required to insert a few instructions in our program. Let us say we have a long program as shown in the following.

Memory address	**Contents**
F800H	LDA FE00H
F803H	MOV B, A
F804H	MOV D, A
.	
.	
F810H	ADD E
F811H	NOP
.	
.	
FD56H	HLT

When the program is executed, let us say we do not get the desired result, because the program is wrong. Then we go about debugging the program in single step mode. Let us say, we come to the conclusion that in between MOV B, A and MOV D, A there should have been MOV C, A also. If we insert the instruction 'MOV C, A' at F804H, then the original program portion from memory location F804H to FD56H should be moved down by one position. This amounts

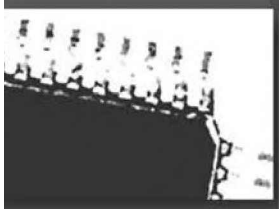
to almost rewriting the whole program, as the error has occurred in the beginning of the program. Anyway, we do not have to lose heart, if we have NOP instructions at several places in our program. Suppose we have a NOP instruction at F811H, then we have to move down only a small portion of the program from F804H to F810H by one location. Now the earlier NOP instruction is replaced with ADD E.

Thus, it is a good programming practice to have a few NOP instructions in the program at regular intervals, especially during program development. Alternatively, the user can make use of 'insert' and 'delete' keys provided on the kit, which help in inserting new instructions to the program, or delete existing instructions.

Summary: NOP (1 byte; NOP; 1 opcode)

1. Differentiate between stack pointer and stack.
2. Why the stack should not be implemented in ROM?
3. Explain with examples, the two basic operations performed on stack contents.
4. What is PSW? Write a 8085 assembly language program to exchange contents of accumulator and flag registers.
5. Write a 8085 assembly language program to exchange contents of PSW and HL.
6. Distinguish between the following pairs of instructions.
   a. XTHL and SPHL,
   b. LXI SP, d16 and SPHL.
7. Explain the utility of NOP instruction.

10 Branch Group of Instructions



- More details about program execution
  - *PC—Program counter*
  - *IR—Instruction register*
  - *W and Z registers*
- Unconditional jump instructions
  - *JMP a16—Unconditional direct jump*
  - *PCHL—Unconditional indirect jump*
- Conditional jump instructions
  - *JNC a16—jump if not Carry*
  - *JC—jump if carry*
  - *JNZ—jump if not zero result*
  - *JZ a16—jump if zero result*
  - *JPO a16—jump if parity odd*
  - *JPE a16—jump if parity even*
  - *JP a16—jump if positive*
  - *JM a16—jump if minus*
- Unconditional call and return instructions
  - *Difference between call and jump instructions*
- Conditional call instructions
  - *CNC a16—call if no carry*
  - *CC a16—call if carry*
  - *CNZ a16—call if not zero result*
  - *CZ a16—call if zero result*
  - *CPO a16—call if parity odd*
  - *CPE a16—call if parity even*
  - *CP a16—call if positive*
  - *CM a16—call if minus*
- Conditional return instructions
  - *RNC—return if not carry*

- *RC—return if carry*
- *RNZ—return if not zero result*
- *RZ—return if zero result*
- *RPO—return if parity odd*
- *RPE—return if parity even*
- *RP—return if positive*
- *RM—return if minus*

■ RSTn—restart instructions

■ Questions

A total of eight instruction types covering 36 instructions will be explained in this chapter.

The first part of the chapter gives more details about program execution. Various instruction types, such as the conditional and unconditional jump instructions, unconditional and conditional call and return instructions are dealt with in the remaining part of the chapter.

■ 10.1 MORE DETAILS ABOUT PROGRAM EXECUTION

The way 8085 executes a simple program (without 'jump' instructions) is as follows. It fetches the first instruction from memory and executes it. Then it fetches the next instruction from memory and executes it. It continues this way till the halt instruction is fetched and executed.

For example, to compute the 2's complement of the number at location F840H and store the result in location F850H, the program shown in Fig. 10.1 has to be executed.

```
LDA F840H
CMA
INR A
STA F850H
HLT
```

Fig. 10.1
Assembly language program to compute 2's complement

Let us say, the machine language program is stored starting from memory location F820H as shown in Fig. 10.2.

Memory address	**Memory contents**
F820H	3AH
F821H	40H
F822H	F8H
F823H	2FH
F824H	3CH
F825H	32H
F826H	50H
F827H	F8H
F828H	76H

Fig. 10.2
Machine language program to compute 2's complement

To execute the program, 8085 will have to send out the address F820H and fetch the opcode 3AH for the first instruction 'LDA F840H' from memory for execution. In such a case, from which register in 8085 the address value F820H is sent out? The address is sent out from a special purpose register called program counter (PC for short).

10.1.1 PC—Program Counter

PC is a 16-bit register. It contains a memory address. Suppose the PC contents are F820H, then it means that the 8085 desires to fetch the instruction byte at F820H. After fetching the byte at F820H, the PC is automatically incremented by 1. This way 8085 becomes ready to fetch the next byte of the instruction (in case instruction fetch is incomplete), or fetch the next opcode (in case instruction fetch is over).

In the previous program example, first of all PC is loaded with the value F820H. This is done by typing the 'Go' key, then typing 'F820', and finally typing the 'Exec' key. Then the 8085 performs the following action. It sends out F820H the address, which is the content of the PC. From location F820H it receives 3AH, the opcode for LDA. It is received in an 8-bit register called instruction register (IR for short), as shown in Fig. 10.3.

Memory address	*Memory contents*	*Received in 8085 by*
F820H	3AH	IR register
F821H	40H	Z register
F822H	F8H	W register
F823H	2FH	IR register
F824H	3CH	IR register
F825H	32H	IR register
F826H	50H	Z register
F827H	F8H	W register
F828H	76H	IR register

Fig. 10.3
Table indicating instruction fetch process

10.1.2 IR—Instruction Register

IR is a special purpose register, which is used to receive the 8-bit opcode portion of an instruction. It is not accessible to the programmer. What it means is that there are no instructions by which the programmer can load it with values of his choice. For example, instructions like 'MOV IR, D' or 'MVI IR, 45H' are not present in the instruction set of 8085. Thus, IR register is not shown in the programmer's view of 8085.

10.1.3 W and Z Registers

After the opcode is fetched from location F820H, the PC is incremented to F821H and is sent out as address. Then 8085 receives 40H, the LS byte of data address F840H. It is received in an 8-bit register called Z register. Z is a register, which is used to receive the LS byte portion of the 16-bit address in a 3-byte instruction. It is not accessible to the programmer. Thus, Z register is not shown in the programmer's view of 8085.

Next, PC is incremented to F822H and it receives F8H, the MS byte of data address F840H. It is received in an 8-bit register called W register. W is a register that is used to receive the MS byte portion of a 16-bit address in an instruction. It is not accessible to the programmer.

Now PC is incremented to F823H, but the next instruction fetch does not start as yet. This is because, so far, 8085 has only completely fetched the instruction code for LDA F840H, but it has not yet executed it!

The 8085 executes the instruction LDA F840H, by sending out the address F840H from W and Z registers, and loading the accumulator with the contents of memory location F840H.

Only after the 8085 executes 'LDA F840H' instruction, it sends out the address F823H, which is the content of PC, and receives 2FH, the opcode for CMA. It is received in the IR register. Meanwhile the PC is incremented to F824H. As the complete instruction is received by the 8085, it then executes it using the ALU for the complement operation.

Only after CMA instruction is executed by the 8085, it sends out the address F824H, which is the content of PC, and receives 3CH, the opcode for INR A. It is received in the IR register. Meanwhile the PC is incremented to F825H. As the complete instruction is received by the 8085, it then executes it using the ALU for the increment operation.

Only after INR A instruction is executed by the 8085, it sends out the address F825H, which is the content of PC, and receives 32H, the opcode for STA. It is received in the IR register. Meanwhile the PC is incremented to F826H.

Then F826H from PC is sent out as address and 8085 receives 50H, the LS byte of address F850H. It is received in Z register. Meanwhile PC is incremented to F827H. Then F827H from PC is sent out as address and 8085 receives F8H, the MS byte of address F850H. It is received in W register. Meanwhile PC is incremented to F828H. But the next instruction fetch at F828H does not start as yet. This is because, so far the 8085 has only completely fetched the instruction code for STA F850H, but it has not yet executed it! The 8085 executes the instruction STA F850H, by sending out the address F850H from W and Z registers, and storing the accumulator contents in memory location F850H.

Only after STA F850H instruction is executed by the 8085, it sends out the address F828H, which is the content of PC, and receives 76H, the opcode for HLT. It is received in the IR register. Meanwhile the PC is incremented to F829H. As the complete instruction is received by the 8085, it then executes it, which results in halting the processor.

The role of PC in 8085 for simple programs (without jump instructions) can be summarized as follows:

PC is a 16-bit register. It contains a memory address. Suppose the PC contents are F820H, then it means that the 8085 desires to fetch and execute the instruction starting at F820H. After fetching the instruction starting at F820H, the PC is automatically incremented by 1, 2, or 3 depending on the instruction length. Then the instruction is executed, during which the PC value is not changed.

This way PC will be pointing to the next instruction by the time the current instruction is executed.

■ 10.2 UNCONDITIONAL JUMP INSTRUCTIONS

In simple programs there is a linear program flow from the first instruction to the last instruction in the program. But the real power of a computer exists in its ability to change program flow, and the programs should be written to take advantage of this ability.

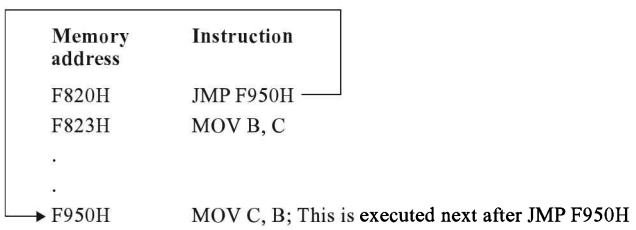
Change in program flow is needed when a sequence of instructions is to be executed repeatedly. It is also needed when it is required to choose between two or more sequences of actions based on some conditions. The branch group of instructions in 8085 effects such a change in program flow. The branch instructions can be used for effecting a forward branch or a backward branch.

An important subgroup of the branch group of instructions is the jump instructions. The jump instructions are classified into:

– Unconditional jump instructions, and
– Conditional jump instructions.

10.2.1 JMP a16—Unconditional Direct Jump

JMP is a mnemonic that stands for '**JuMP**' and 'a16' stands for any 16-bit address. This instruction is used to jump to the address a16 provided in the instruction. 'JMP F950H' is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.



In the previous example, after 8085 fetches 'JMP F950H' the PC value would have been automatically incremented by 3 to F823H. Functionally, JMP F950H can be treated as:

JMP F950H = LXI PC, F950H

Thus, execution of JMP F950H results in loading of PC with the value F950H, overwriting the earlier value of F823H. Hence, after execution of JMP F950H, the instruction MOV C, B at memory location F950H will be fetched and executed. In this example, a forward jump was effected. JMP F805H instruction at location F820H, effects a backward jump.

In the instruction set of 8085, 'JMP' is used instead of 'LXI PC', because JMP very explicitly indicates a change in program flow. But, LXI PC just indicates that PC value is changed, but its implied change in program flow may go unnoticed by the programmer.

The result of execution of 'JMP F950H' can be indicated in terms of change in PC value as follows.

	Before	***After***
(PC)	F820H	F950H

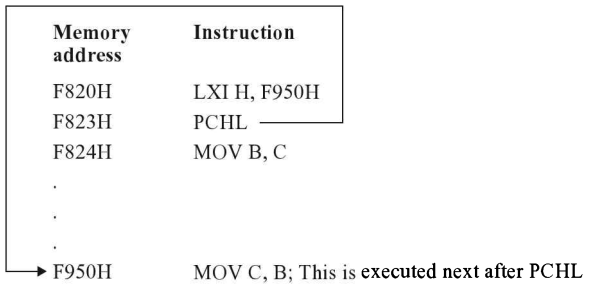
Summary: JMP a16 (3 bytes; JMP F950H; 1 opcode)

10.2.2 PCHL—Unconditional Indirect Jump

PCHL is a mnemonic, which stands for 'Load **PC** with contents of **HL**'. This instruction is used to jump to the address provided in HL register pair. Thus it is an unconditional indirect jump instruction. Because of its indirect jump feature, it is not very commonly used.

It is a 1-byte instruction compared with the direct jump instruction, which is 3-bytes long. Because of this size advantage, it can be useful for jumping to a frequently used portion of the program.

The result of execution of this instruction is shown below with an example.



In the previous example, after 8085 fetches 'PCHL' the PC value would have been automatically incremented by 1 to F824H.

But execution of PCHL results in loading of PC with the value F950H, which is the content of HL, overwriting the earlier value of F824H. Hence, after execution of PCHL, the instruction 'MOV C, B' at memory location F950H will be fetched and executed.

The result of execution of 'PCHL' can be indicated in terms of change in PC value as follows.

	Before	***After***
(HL)	F950H	
(PC)	F823H	F950H

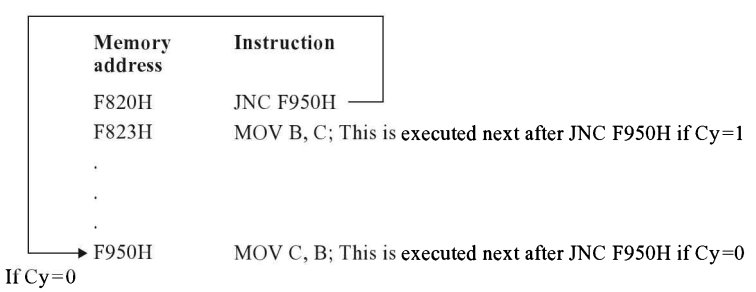
Summary: PCHL (1 byte; PCHL; 1 opcode)

■ 10.3 CONDITIONAL JUMP INSTRUCTIONS

The conditional jump instructions of 8085 perform a jump based on the value of a single flag. The jump takes place based on the value of carry flag, zero flag, parity flag, or sign flag. There is no jump instruction based on the value of auxiliary flag. This is because, generally no one is interested in performing a jump based on this flag!

10.3.1 JNC a16—JUMP IF NOT CARRY

JNC is a mnemonic, which stands for '**J**ump if **N**ot **C**arry', and 'a16' stands for any 16-bit address. This instruction is used to jump to the address a16 provided in the instruction, only if carry flag value is 0. If carry flag value is 1, program flow continues sequentially. 'JNC F950H' is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.



In the previous example, after 8085 fetches 'JNC F950H' the PC value would have been automatically incremented by 3 to F823H. But JNC F950H instruction execution results in loading of PC with the value F950H, only if Cy flag value is 0.

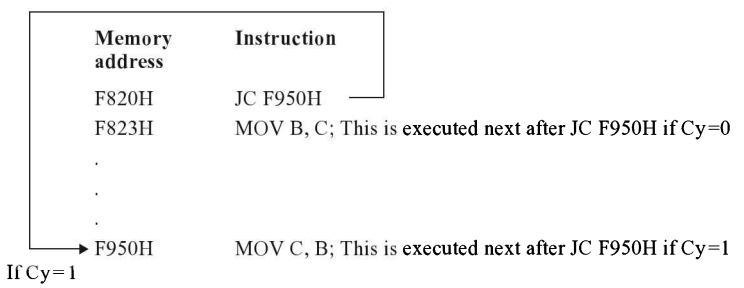
Hence, after execution of JNC F950H, the instruction 'MOV C, B' at memory location F950H will be fetched and executed only if Cy flag value is 0. If Cy flag value is 1, the PC value will remain as F823H, and so the instruction 'MOV B, C' will be executed.

Summary: JNC a16 (3 bytes; JNC F950H; 1 opcode)

10.3.2 JC a16—Jump if Carry

JC is a mnemonic, which stands for '**J**ump if **C**arry'. This instruction is used to jump to the address a16 provided in the instruction, only if carry flag value is 1. If carry flag value is 0, program flow continues sequentially.

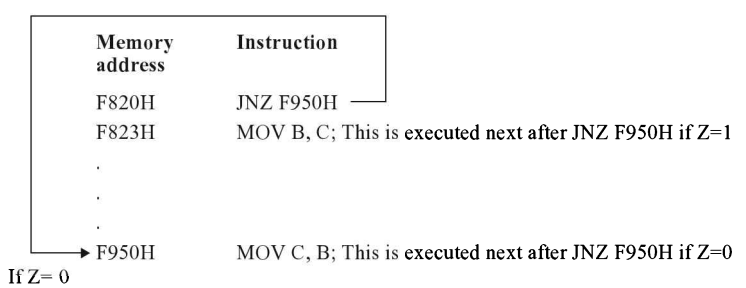
'JC F950H' is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.



Summary: JC a16 (3 bytes; JC F950H; 1 opcode)

10.3.3 JNZ a16—Jump if Not Zero Result

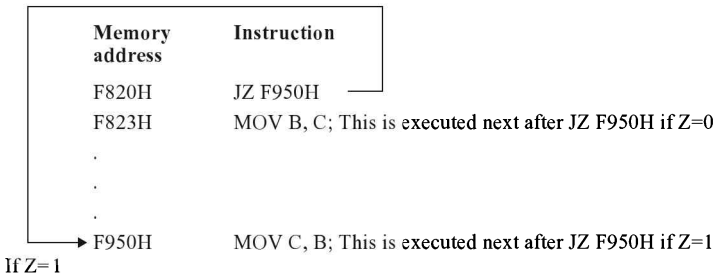
JNZ is a mnemonic, which stands for '**J**ump if **N**ot **Z**ero result'. This instruction is used to jump to the address a16 provided in the instruction, only if result is not zero, indicated by Z flag value of 0. If Z flag value is 1, program flow continues sequentially. 'JNZ F950H' is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.



Summary: JNZ a16 (3 bytes; JNZ F950H; 1 opcode)

10.3.4 JZ a16—Jump if Zero Result

JZ is a mnemonic, which stands for '**J**ump if **Z**ero result'. This instruction is used to jump to the address a16 provided in the instruction, only if result is zero, indicated by Z flag value of 1. If Z flag value is 0, program flow continues sequentially. 'JZ F950H' is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.



Summary: JZ a16 (3 bytes; JZ F950H; 1 opcode)

10.3.5 JPO a16—Jump if Parity Odd

JPO is a mnemonic, which stands for '**J**ump if **P**arity **O**dd'. This instruction is used to jump to the address a16 provided in the instruction, only if parity flag value is 0. If parity flag value is 1, program flow continues sequentially. 'JPO F950H' is an example instruction of this type. It is a 3-byte instruction.

Summary: JPO a16 (3 bytes; JPO F950H; 1 opcode)

10.3.6 JPE a16—Jump if Parity Even

JPE is a mnemonic, which stands for '**J**ump if **P**arity **E**ven'. This instruction is used to jump to the address a16 provided in the instruction, only if parity flag value is 1. If parity flag value is 0, program flow continues sequentially.

Summary: JPE a16 (3 bytes; JPE F950H; 1 opcode)

10.3.7 JP a16—Jump if Positive

JP is a mnemonic, which stands for '**J**ump if **P**ositive'. This instruction is used to jump to the address a16 provided in the instruction, only if sign flag value is 0. If sign flag value is 1, program flow continues sequentially.

Summary: JP a16 (3 bytes; JP F950H; 1 opcode)

10.3.8 JM a16—Jump if Minus

JM is a mnemonic, which stands for '**J**ump if **M**inus'. This instruction is used to jump to the address a16 provided in the instruction, only if sign flag value is 1. If sign flag value is 0, program flow continues sequentially.

Summary: JM a16 (3 bytes; JM F950H; 1 opcode)

■ 10.4 UNCONDITIONAL CALL AND RETURN INSTRUCTIONS

Very often a particular sequence of instructions have to be used at several points in the program. Writing the same sequence of instructions at these various points can be avoided by writing this series of instructions as a subprogram. Such subprograms are also variously termed as subroutines, or procedures.

Whenever the instructions in a subroutine are required to be executed, we branch to the subroutine using the CALL instruction. It is a 3-byte instruction, with 1 byte for the opcode, and 2 bytes for the address of subroutine. CALL stands for 'call a subroutine'. After executing the instructions in the subroutine we return to the next instruction after the CALL instruction in the main program. This is achieved by the execution of the RET (stands for **RET**urn from subroutine) instruction in the subroutine. RET is a 1-byte instruction.

10.4.1 Difference between Call and Jump Instructions

The difference between a JMP instruction and a CALL instruction is as follows. If a JMP instruction is executed, we jump to the destination location, and the execution carries on from there, without bothering to come back later to the instruction after the JMP.

If a CALL instruction is executed, we jump to the subroutine, and the execution carries on from there till the RET instruction is executed in the subroutine, and then we come back to the instruction after the CALL in the main program.

The address of the next instruction after the CALL instruction is called the return address. This is the address to which the program flow returns when the RET instruction is executed by the 8085. By the time the 8085 fetches the call instruction, the PC would have been incremented by 3, and will be pointing to the next instruction after call. In other words, PC will have the return address by the time the call instruction is fetched.

In order to facilitate such a return, the CALL instruction will first of all store above the top of stack, the return address. Only then the branch to the subroutine takes place.

Thus a CALL instruction can be visualized as push PC value on the stack, and then branch to the subroutine. Thus CALL F900H can be treated as:

CALL F900H = PUSH PC + JMP F900H

Note that there is no direct instruction like PUSH PC in 8085. As part of the execution of CALL instruction, PUSH PC operation is performed.

The RET instruction, which is at the end of a subroutine, loads the PC with the return address popped out from the top of the stack, so that a branch back to the calling program at the next instruction after CALL is achieved. Thus RET can be treated as:

RET = POP PC

In the instruction set of 8085, 'RET' is used instead of 'POP PC', because RET very explicitly indicates a return back to the calling program. But, POP PC just indicates that PC value is changed, but its implied return to the calling program may go unnoticed by the programmer.

The use of CALL F950H to branch to the subroutine at F950H, and the RET instruction in the subroutine to get back to the main program are illustrated in Fig. 10.4.

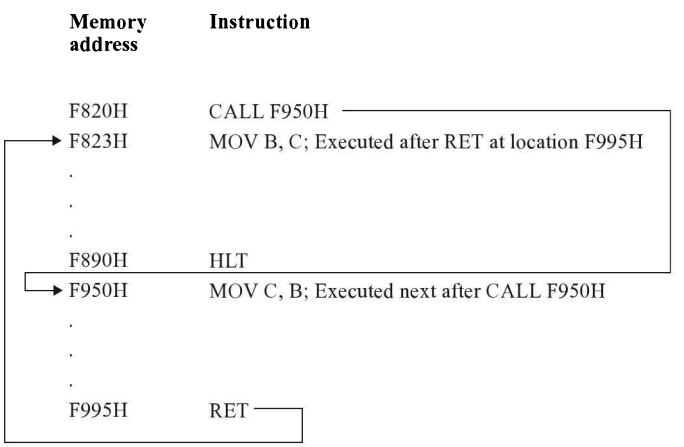


Fig. 10.4
Usage of CALL and RET instructions

The way stack contents get affected due to execution of 'CALL F950H' is shown below.

	Before	*After*
(PC)	F820H	F950H
(SP)	FC08H	FC06H
(FC07)	60H	F8H
(FC06)	70H	23H

Note that if the CALL instruction is at location F820H, the return address stored on the top of the stack is F823H.

Summary: CALL a16 (3 bytes; CALL F950H; 1 opcode)

The way in which the stack contents get affected due to execution of RET is shown below.

	Before	*After*
(PC)	F995H	F823H
(SP)	FC06H	FC08H
(FC06)	23H	
(FC07)	F8H	

Summary: RET (1 byte; RET; 1 opcode)

Using this approach, we can branch any number of times to the subroutine, which is written only once. This saves a lot of memory space for the complete program.

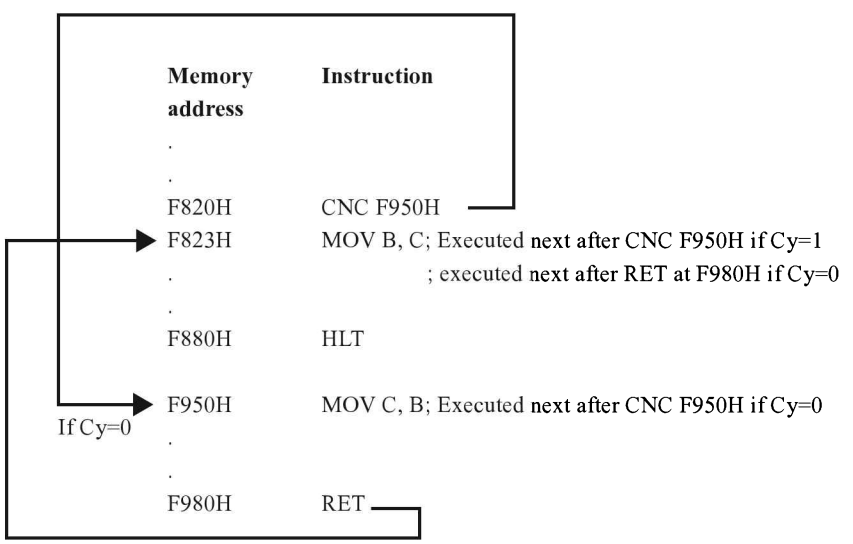
Also usage of subroutines helps in modular design for solving a programming problem. It makes the main program look very simple and compact.

■ 10.5 CONDITIONAL CALL INSTRUCTIONS

The conditional call instructions of 8085 branch to a subroutine based on the value of a single flag. The branch takes place based on the value of Cy flag, Z flag, P flag, or S flag. There is no call instruction based on the value of auxiliary flag. This is because, generally no one is interested in branching to a subroutine based on this flag! The conditional call instructions are 3 bytes in length, 1 byte for the opcode, and another 2 bytes for the subroutine address.

10.5.1 CNC a16—CALL IF NOT CARRY

CNC is a mnemonic, which stands for '**C**all if **N**ot **C**arry'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if Cy flag value is 0. If Cy flag value is 1, program flow continues in the main program sequentially. 'CNC F950H' is an example instruction of this type. It is a 3-byte instruction. The result of execution of this instruction is shown below with an example.



In the previous example, after 8085 fetches 'CNC F950H' the PC value would have been automatically incremented by 3 to F823H. But CNC F950H instruction execution results in saving PC value on the stack and loading of PC with the value F950H, only if Cy flag value is 0.

Hence, after execution of CNC F950H, branch to subroutine at memory location F950H will take place only if Cy flag value is 0. After executing the RET instruction in the subroutine, the program flow will continue with the instruction at F823H in the main program.

If Cy flag value is 1, the PC value will remain as F823H, and so the program flow continues with the instruction MOV B, C in the main program.

Summary: CNC a16 (3 bytes; CNC F950H; 1 opcode)

10.5.2 CC a16—Call if Carry

CC is a mnemonic, which stands for '**C**all if **C**arry'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if Cy flag value is 1. If Cy flag value is 0, program flow continues in the main program sequentially.

Summary: CC a16 (3 bytes; CC F950H; 1 opcode)

10.5.3 CNZ a16—Call if Not Zero Result

CNZ is a mnemonic, which stands for '**C**all if **N**ot **Z**ero result'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if result is not zero, indicated by Z flag value of 0. If Z flag value is 1, program flow continues in the main program sequentially.

Summary: CNZ a16 (3 bytes; CNZ F950H; 1 opcode)

10.5.4 CZ a16—Call if Zero Result

CZ is a mnemonic, which stands for '**C**all if **Z**ero result'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if result is zero, indicated by Z flag value of 1. If Z flag value is 0, program flow continues in the main program sequentially.

Summary: CZ a16 (3 bytes; CZ F950H; 1 opcode)

10.5.5 CPO a16—Call if Parity Odd

CPO is a mnemonic, which stands for '**C**all if **P**arity **O**dd'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if P flag value is 0. If P flag value is 1, program flow continues in the main program sequentially.

Summary: CPO a16 (3 bytes; CPO F950H; 1 opcode)

10.5.6 CPE a16—Call if Parity Even

CPE is a mnemonic, which stands for '**C**all if **P**arity **E**ven'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if P flag value is 1. If P flag value is 0, program flow continues in the main program sequentially.

Summary: CPE a16 (3 bytes; CPE F950H; 1 opcode)

10.5.7 CP a16—Call if Positive

CP is a mnemonic, which stands for '**C**all if **P**ositive'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if S flag value is 0. If S flag value is 1, program flow continues in the main program sequentially.

Summary: CP a16 (3 bytes; CP F950H; 1 opcode)

10.5.8 CM a16—CALL IF MINUS

CM is a mnemonic, which stands for '**C**all if **M**inus'. This instruction is used to branch to the subroutine whose 16-bit address is provided in the instruction, only if S flag value is 1. If S flag value is 0, program flow continues in the main program sequentially.

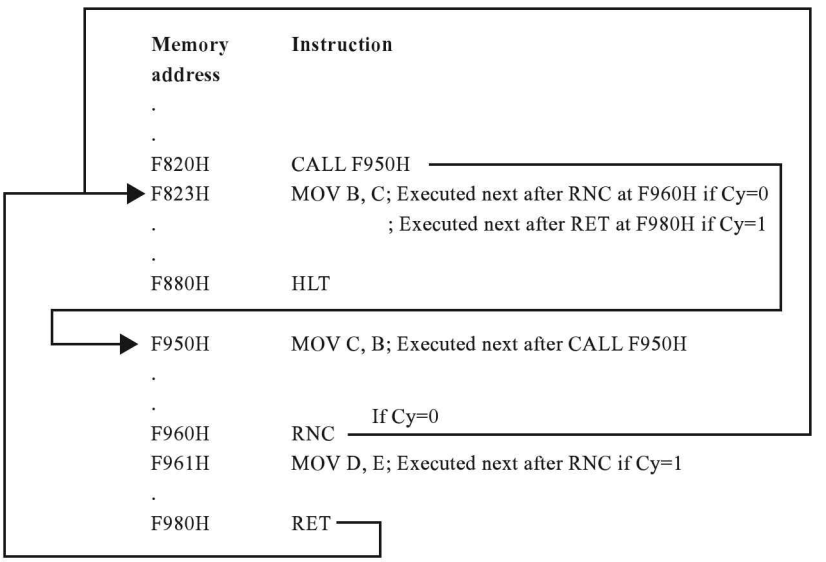
Summary: CM a16 (3 bytes; CM F950H; 1 opcode)

■ 10.6 CONDITIONAL RETURN INSTRUCTIONS

The conditional return instructions of 8085 effect a return to the main program based on the value of a single flag. The return takes place based on the value of Cy flag, Z flag, P flag, or S flag. There is no return instruction based on the value of AC flag. This is because, generally no one is interested in returning to the main program based on this flag! The conditional return instructions are 1 byte in length.

10.6.1 RNC—RETURN IF NOT CARRY

RNC is a mnemonic, which stands for '**R**eturn if **N**ot **C**arry'. This instruction is used to return to the main program, only if Cy flag value is 0. If Cy flag value is 1, program flow continues in the subroutine sequentially. It is a 1-byte instruction. The result of execution of this instruction is shown below with an example.



In the previous example, when 8085 executes CALL F950H instruction, it saves PC value of F823H on the top of the stack, and branches to the subroutine at F950H.

In the subroutine, after 8085 fetches 'RNC' the PC value would have been automatically incremented by 1 to F961H. But RNC instruction execution results in loading PC value with the return address from the stack top, only if Cy flag value is 0.

Hence, after execution of RNC, return to the main program at memory location F823H will take place only if Cy flag value is 0.

If Cy flag value is 1, the PC value will remain as F961H, and so the program flow continues with the instruction MOV D, E in the subroutine. Finally when the 8085 executes the RET instruction in the subroutine, return to the main program at location F823H takes place.

Summary: RNC (1 byte; RNC; 1 opcode)

10.6.2 RC—Return if Carry

RC is a mnemonic, which stands for '**R**eturn if **C**arry'. This instruction is used to return to the main program, only if Cy flag value is 1. If Cy flag value is 0, program flow continues in the subroutine sequentially.

Summary: RC (1 byte; RC; 1 opcode)

10.6.3 RNZ—Return if Not Zero Result

RNZ is a mnemonic, which stands for '**R**eturn if **N**ot **Z**ero result'. This instruction is used to return to the main program, only if result is not zero, indicated by Z flag value of 0. If Z flag value is 1, program flow continues in the subroutine sequentially.

Summary: RNZ (1 byte; RNZ; 1 opcode)

10.6.4 RZ—Return if Zero Result

RZ is a mnemonic, which stands for '**R**eturn if **Z**ero result'. This instruction is used to return to the main program, only if result is 0, indicated by Z flag value of 1. If Z flag value is 0, program flow continues in the subroutine sequentially.

Summary: RZ (1 byte; RZ; 1 opcode)

10.6.5 RPO—Return if Parity Odd

RPO is a mnemonic, which stands for '**R**eturn if **P**arity is **O**dd'. This instruction is used to return to the main program, only if P flag value is 0. If P flag value is 1, program flow continues in the subroutine sequentially.

Summary: RPO (1 byte; RPO; 1 opcode)

10.6.6 RPE—Return if Parity Even

RPE is a mnemonic, which stands for '**R**eturn if **P**arity is **E**ven'. This instruction is used to return to the main program, only if P flag value is 1. If P flag value is 0, program flow continues in the subroutine sequentially.

Summary: RPE (1 byte; RPE; 1 opcode)

10.6.7 RP—Return if Positive

RP is a mnemonic, which stands for '**R**eturn if **P**ositive'. This instruction is used to return to the main program, only if S flag value is 0. If S flag value is 1, program flow continues in the subroutine sequentially.

Summary: RP (1 byte; RP; 1 opcode)

10.6.8 RM—Return if Minus

RM is a mnemonic, which stands for '**R**eturn if **M**inus'. This instruction is used to return to the main program, only if S flag value is 1. If S flag value is 0, program flow continues in the subroutine sequentially.

Summary: RM (1 byte; RM; 1 opcode)

■ 10.7 RST*n*—RESTART INSTRUCTIONS

In this case *n* has a value from 0 to 7 only. Thus the eight possible RST instructions are RST 0, RST 1, ... , RST 7. These are basically single-byte call instructions. Functionally RST *n* instruction is:

$$\text{RST } n = \text{CALL } n * 8$$

For example, RST 2 is functionally equivalent to CALL 2 * 8 = CALL 0010H. The advantage of RST 2 is that it is only 1 byte, whereas CALL 0010H is 3-byte long. Thus RST instructions are useful for branching to frequently used subroutines.

The reader may wonder how RST *n* is only 1-byte long, whereas MVI B, d8 is 2-byte long. The reason is that *n* value is restricted to the range 0–7. So only 3 bits are needed to denote *n* value, and the other 5 bits in the byte provide the code for RST.

RST 2 is an example instruction of this type. It is a 1-byte instruction. It is functionally same as CALL 0010H = PUSH PC + JMP 0010H. It causes a branch to subroutine at 0010H. Similarly, RST 3 causes a branch to subroutine at 3 * 8 = 0018H. Thus, the subroutine, which starts at location 0010H should not go beyond memory location 0017H. So at the most only eight locations are available for the subroutine, which is too small in general. This limitation is overcome by branching to a subroutine at some other memory location, like F950H. It is achieved by the combination of RST 2 instruction, and JMP F950H instruction at memory location 0010H, as shown in Fig. 10.5.

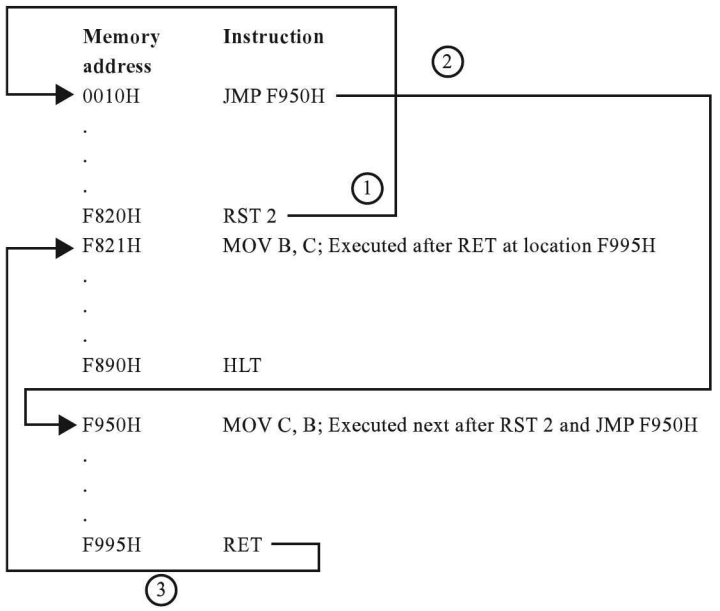


Fig. 10.5
Use of RST 2 to branch to subroutine at F950H

The way in which the stack contents get affected due to execution of RST 2 is shown below.

	Before	*After*
(PC)	F820H	0010H
(SP)	FC08H	FC06H
(FC07)	60H	F8H
(FC06)	70H	21H

Note that if the RST 2 instruction is at location F820H, the return address stored on the top of the stack is F821H. Thus PC value is stored on the top of the stack, and a branch to 0010H takes place. But then, because of the JMP F950H at location 0010H, branch to subroutine at F950H finally takes place.

Many times, an I/O port supplies an RST instruction to 8085, when the 8085 is interrupted on INTR pin. Interrupts are discussed in a later chapter.

'RST' stands for 'restart'. They are called restart instructions because, generally one of these instructions is going to transfer control to the monitor program in the microprocessor kit. When a microprocessor kit is switched off, and then switched on, the control is transferred to the monitor program. Thus the action performed by an RST instruction is similar to restarting the kit, and hence the name 'restart' instruction. On the ALS-SDA-85M kit RST 1 instruction transfers control to the monitor program.

Summary: RST *n* (1 byte; RST 5; 8 opcodes)

The complete role of PC in 8085 can be summarized as follows:

PC is a 16-bit register. It contains a memory address. Suppose the PC contents are F820H. Then it means that the 8085 desires to fetch and execute the instruction starting at F820H. After fetching the instruction starting at F820H, the PC is automatically incremented by 1, 2, or 3 depending on the instruction length. Then the instruction is executed, during which the PC value is not

changed, if the instruction did not belong to the branch group. This way PC will be pointing to the next instruction by the time the current instruction is executed.

If the instruction belonged to the branch group the following actions take place.

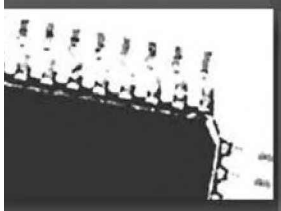
- If the current instruction is an unconditional direct jump or call instruction, PC is loaded with the value provided in the JMP or CALL instruction. Thus, a jump is effected.
- If the current instruction is PCHL, PC is loaded with the value present in HL register pair. Thus, a jump is effected.
- If the current instruction is an RST *n* instruction, PC is loaded with the value *n**8. Thus, a jump is effected.
- If the current instruction is a conditional jump or call instruction, then
  - PC is loaded with the value provided in the conditional jump or call instruction, if the condition is satisfied. Thus, a jump is effected.
  - PC value is not altered if the condition is not satisfied. In such a case, there will not be a jump.

1. Explain the role of Program Counter.
2. What is the purpose of Instruction register?
3. What is the function of W and Z registers?
4. Describe the two types of unconditional jump instructions.
5. List the conditional jump instructions of 8085, and explain any one instruction.
6. What are subroutines? How are they useful?
7. Describe the working of the instructions CALL and RET.
8. Compare JMP and CALL instructions.
9. List the conditional call instructions of 8085, and explain any one instruction.
10. List the conditional return instructions of 8085, and explain any one instruction.
11. Explain the working of RST instructions and their use.
12. Distinguish between the following pairs of instructions:
    a. RST 5 and CALL 0028H;
    b. JMP F090H and CALL F090H;
    c. SPHL and PCHL;
    d. RET and RPO;
    e. CALL F090H and CM F090H.
13. RST 5 is 1-byte long, and MVI B, 25H is 2-bytes long. Justify with reason.
14. It is desired to branch to subroutine at location F960H using RST 4 instruction. Explain how you achieve it.
15. Write a 8085 assembly language program, which takes the 8-bit data from memory location X, and counts the number of 1s and 0s in this byte, and stores the result in memory locations X + 1 and X + 2, respectively.

16. Write a 8085 assembly language program, which performs the addition of two 8-bit signed numbers at locations X and X + 1 respectively. If the answer is correct, store it in memory location X + 2. If the answer is wrong, store it in location Y.

    (Note: Addition of 40H and 50H gives the wrong answer 90H. It is a wrong answer because 40H and 50H are positive, while 90H is negative.)

17. Write a 8085 assembly language program to generate all Fibonacci numbers, which can be represented using 8 bits, and store them in successive locations starting from location X.



11 Chip Select Logic

- Concept of chip selection
- RAM chip—Pin details and its address range
- Multiple memory address range
- Working of 74138 decoder IC
- Use of 74138 to generate chip select logic
  - *Advantage of multiple chip select lines*
- Use of 74138 in ALS-SDA-85M kit
- Questions

In the previous chapters a total of 59 instruction types had been discussed. Only seven more types are still to be discussed. However, let us have a digression and discuss about the concept of chip selection.

11.1 CONCEPT OF CHIP SELECTION

In a microcomputer system, the microprocessor is the master, which controls all the operations of the computer. This is because, it is the microprocessor in which the control unit is embedded. In addition to the processor, there may be a number of RAM chips, EPROM chips, I/O port chips, and other special peripheral chips like 8253 timer, etc.

But at any instant of time, the processor can communicate with only one chip. The processor selects the chip with which it would like to communicate by activating chip select pin/pins of the chip. Thus, generally all chips, other than the processor itself, will have one or more chip select input pins. A chip select pin can be active low or active high. It is the choice of the chip designer. User has no choice in this respect. Only when the chip select pins of the chip are activated, the chip is selected for communication with the processor. A chip select line is also variously called as 'device select line', 'chip enable line', or 'device enable line'.

For example, 74138 has three chip select pins as described in a later section. The 8255 PPI chip to be discussed in a later chapter has only one chip select line. It is advantageous sometimes to have more than one chip select pin, as explained in a later section.

■ 11.2 RAM CHIP—PIN DETAILS AND ADDRESS RANGE

Let us say, we have a 2K × 8 RAM chip. Here, 2K × 8 means that there are 2K (= 2 × 1,024 = 2,048) memory locations, with each location having 8 bits of information. It is available as a 24-pin DIP IC. Its functional pin diagram is as shown in Fig. 11.1.

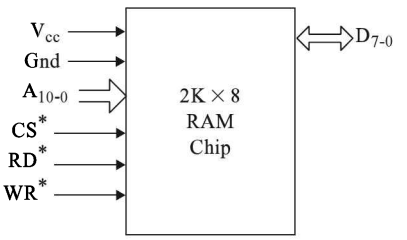


Fig. 11.1
Functional pin diagram of
2K × 8 RAM chip

Note that there are 11 address input pins to select one of the 2K = 2^{11} memory locations in the chip. There are eight bi-directional pins for data transfer with the processor. The action desired by the processor is indicated by the RD* and WR* input pins to the RAM chip. There are two pins meant for power supply connection of +5 V DC and Ground. Then there is an active low chip select pin.

Now the question is what is the starting address for this chip? Does it have to always start from 0000H, or can it start at say, C000H? If we think that the address should always start from 0000H, it is not convenient for the following reason. Suppose we had a 2K × 8 RAM chip in our system, with the address starting at 0000H. Sometime later, we decide to increase the RAM capacity in the system by another 2K × 8. If this added memory also starts with the address 0000H, effectively we have not made any addition to the RAM capacity at all! To be useful, the added memory address range should not overlap with the existing address range. Thus, it should be possible for the designer of the computer system to place the memory in the address range of his choice. Of course, in 8085-based system it should be within the address range 0000H–FFFFH.

Let us say it is desired to place the 2K × 8 RAM such that its starting address is C000H. One possible circuit connection is as shown in Fig. 11.2.

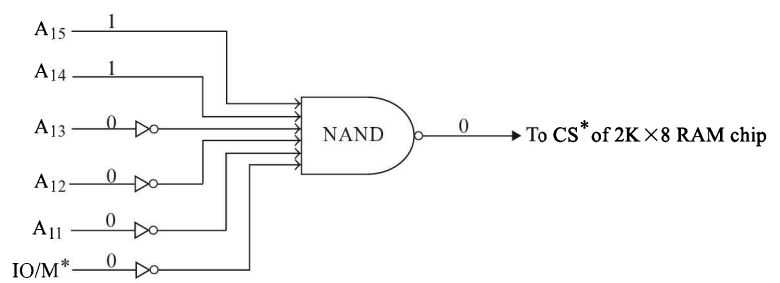


Fig. 11.2
Chip selection
circuit to provide
2K × 8 RAM with
starting address
as C000H

Note that there are 11 address input pins to select one of the 2K = 2^{11} memory locations in the 2K × 8 RAM chip. But 8085 sends out a 16-bit address. So what happens to the remaining five lines of address information? These address lines are used for selecting the chip such that the starting address for the RAM becomes C000H as shown in the following.

A_{15-11} selects RAM	A_{10-0} selects a location in RAM
1 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 = C000H

All of the the NAND gate inputs become 1, when A_{15} A_{14} A_{13} A_{12} A_{11} IO/M* = 1 1 0 0 0 0. So the NAND gate output becomes 0, thus activating the chip select pin of the RAM. Thus location 0 in the RAM is addressed by the 8085 as location C000H. Location 1 in this RAM will be addressed as C001H. The last location in this RAM is selected when the LS 11 bits are 111 1111 1111, and as the RAM has to be selected, the MS 5 bits have to be 11000. Thus the last address in this RAM is 11000 111 1111 1111 = C7FFH. Hence, for the chip select circuit shown in Fig. 11.2 the address range will be from C000H–C7FFH.

If the 8085 sends out an address that is outside the range of C000H–C7FFH, the chip does not get selected at all. Suppose the address sent out by 8085 is BFFFH, then the MS 5 bits of address will be 1 0 1 1 1. Hence, the NAND gate output will be 1, and so the chip is not selected. Similarly, suppose the address sent out by 8085 is C800H, then the MS 5 bits of address will be 1 1 0 0 1. Hence, the NAND gate output will be 1, and so the chip is not selected.

It can be easily seen that C000H–C7FFH is actually 2K locations, as follows. If we have memory from C000H to C003H, the memory locations are C000H, C001H, C002H, and C003H, a total of four memory locations. Thus in general, if memory address range is from X to Y, number of memory locations = (Y − X) + 1. Hence number of memory locations in the address range C000H–C7FFH is 7FFH + 1 = 800H = 2048 = 2K.

Note that there is nothing like a unique chip select circuit. The chip select circuit shown in Fig. 11.3 also serves to give the same address range, C000H–C7FFH.

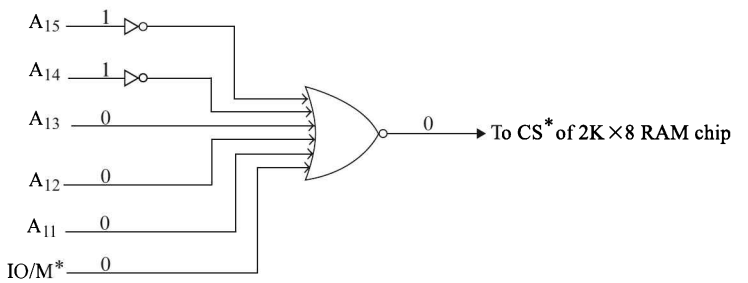


Fig. 11.3 Another equivalent chip select circuit

All the OR gate inputs become 0, when A_{15} A_{14} A_{13} A_{12} A_{11} IO/M* = 1 1 0 0 0 0. So the OR gate output becomes 0, thus activating the chip select pin of the RAM.

Another thing to note is that the starting address should be exactly divisible by the size of the memory chip. For example, a 2K × 8 RAM chip can have any starting address that is divisible by 800H = 2K. Thus, possible starting addresses for a 2K × 8 RAM are 0000H, 0800H, 1000H, ..., F800H. Let us see why it is not possible to have the starting address as, say, C400H for a 2K × 8 RAM.

$A_{15\text{-}11}$ selects RAM	$A_{10\text{-}0}$ selects a location in RAM
1 1 0 0 0	1 0 0 0 0 0 0 0 0 0 = C400H

In this case, the RAM chip is selected, but when address C400H is sent out, location 400H within the chip is selected. Thus, the starting address for a 2K × 8 RAM cannot be C400H.

■ 11.3 MULTIPLE MEMORY ADDRESS RANGE

Suppose we have the chip select circuit as shown in Fig. 11.4, then what is the address range for the RAM?

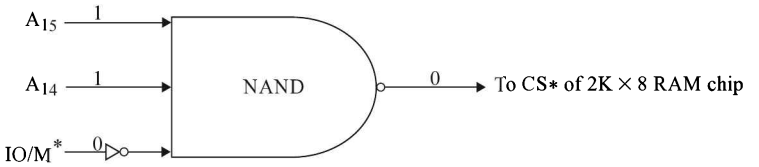


Fig. 11.4
Chip selection that results in multiple address range

In this case, A_{15} and A_{14} are used in selecting the RAM, and $A_{10\text{-}0}$ selects a location in RAM. But A_{13}, A_{12}, and A_{11} can have any value. They have no role to play in chip selection, or selecting a location in the RAM. Thus they are don't cares denoted as '*x*'.

$A_{15\text{-}14}$ selects RAM	$A_{13\text{-}11}$ are don't cares	$A_{10\text{-}0}$ selects a location in RAM
1 1	*x x x*	0 0 0 0 0 0 0 0 0 0

There are eight possible values for $A_{13\text{-}11}$, and as such we will have eight different address ranges possible for the RAM. The address range for the RAM for different values of $A_{13\text{-}11}$ will be as shown in the following table.

$A_{13\text{-}11}$	***Range for RAM***
0 0 0	C000H–C7FFH
0 0 1	C800H–CFFFH
0 1 0	D000H–D7FFH
0 1 1	D800H–DFFFH
1 0 0	E000H–E7FFH
1 0 1	E800H–EFFFH
1 1 0	F000H–F7FFH
1 1 1	F800H–FFFFH

Thus location 0 in this 2K × 8 RAM can be addressed by the 8085 as any of the eight memory locations C000H, C800H, D000H, D800H, E000H, E800H, F000H, F800H. In other words, every location has multiple addresses. This kind of chip selection where we have some don't care values for address lines is called *partially decoded addressing*.

Its disadvantage is that just 2K locations of physical memory have occupied 16K locations of address space. In the previous example, the RAM has occupied locations C000H–FFFFH, a total of 16K locations.

But the advantage of partially decoded addressing is that the chip select circuit is quite simple.

If all the address lines are used for selecting a memory chip, and for selecting a location in that memory chip, it is called 'fully decoded addressing'. Its advantage is that 2K locations of physical memory occupy only 2K locations of address space. For the chip select circuit of Figs. 11.2 or 11.3, the RAM occupies locations C000H–C7FFH, a total of 2K locations. But the disadvantage of fully decoded addressing is that the chip select circuit is quite complex.

■ 11.4 WORKING OF 74138 DECODER IC

As an example of decoder IC, let us consider the popular 3 to 8 decoder IC 74138. It is a 16-pin IC, with its pin out as shown in Fig. 11.5 and functional pin diagram as shown in Fig. 11.6.

As shown in Fig. 11.6, it has three active high input pins denoted as I_2, I_1, and I_0. It has eight active low outputs denoted as O_7*, O_6*, ... , O_0*. The chip needs a power supply of +5 V DC and Ground.

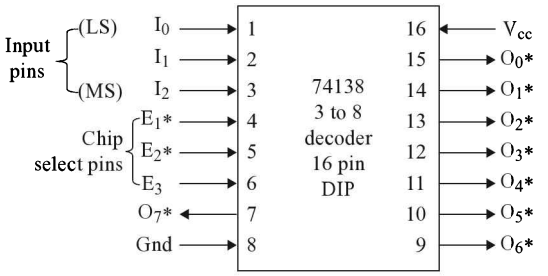


Fig. 11.5
Pin diagram of 74138

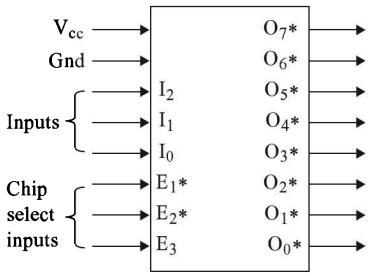


Fig. 11.6
Functional pin diagram of 74138

Depending on the inputs I_2, I_1, and I_0, only one of the output lines is activated. For example, if $I_2\ I_1\ I_0 = 1\ 1\ 1$, then O_7* output line becomes 0, or technically speaking, O_7* line is activated. If $I_2\ I_1\ I_0 = 0\ 1\ 0$, then O_2* output line becomes 0. However, an output line is activated depending on the input, only if the 74138 chip is selected. The 74138 chip is selected when E_1* = 0, E_2* = 0, and E_3 = 1. If this condition is not satisfied then all the output lines will be in the 1 state. In other words, if the chip is selected, then depending on the inputs I_2, I_1, and I_0, one of the output lines will become 0.

■ 11.5 USE OF 74138 TO GENERATE CHIP SELECT LOGIC

Chip 74138 can be used to generate chip select signals for upto eight chips in a microcomputer system. As an example, suppose we have eight numbers of 1K × 8 EPROM chips, and we want the starting address for these chips as 2000H, 2400H, 2800H, ... , 3C00H, respectively. Then the simple interface circuit shown in Fig. 11.7 is adequate.

In Fig. 11.7, 74138 is selected when $A_{15}\ A_{14}\ A_{13} = 0\ 0\ 1$. EPROM-0 is selected when 74138 is selected and $A_{12}\ A_{11}\ A_{10} = 0\ 0\ 0$. The address lines $A_{9\text{-}0}$ select a location in the EPROM-0 after 74138 and EPROM-0 are selected. Thus the starting address for EPROM-0 is:

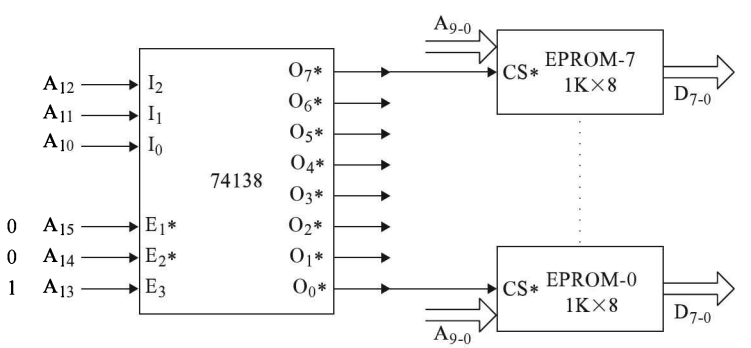


Fig. 11.7
Use of 74138 to generate chip select signals

A_{15-13} selects 74138	A_{12-10} selects an EPROM	A_{9-0} selects a location in EPROM
0 0 1	0 0 0	0 0 0 0 0 0 0 0 0 0 = 2000H

Similarly, EPROM-7 is selected when 74138 is selected and $A_{12}\,A_{11}\,A_{10}$ = 1 1 1. The address lines A_{9-0} select a location in the EPROM-7 after 74138 and EPROM-7 are selected. Thus the starting address for EPROM-7 is:

A_{15-13} selects 74138	A_{12-10} selects an EPROM	A_{9-0} selects a location in EPROM
0 0 1	1 1 1	0 0 0 0 0 0 0 0 0 0 = 3C00H

If we do not use 74138, then we must have separate chip select circuits for each of the eight chips, which will be more expensive as well as more complex.

11.5.1 Advantage of Multiple Chip Select Lines

Suppose we want the EPROMs to have the starting addresses as 4000H, 4400H, ... , 5C00H. 4000H in binary is 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0. In such a case, the 74138 has to be selected for the condition A_{15} $A_{14}\,A_{13}$ = 0 1 0. This can be achieved by the connection shown in Fig. 11.8, which uses two invert gates.

However, the chip selection of 74138 for the condition $A_{15}\,A_{14}\,A_{13}$ = 0 1 0 can be done without using any gates as shown in Fig. 11.9. Here, advantage is taken of the multiple chip select pins of 74138 to simplify its chip selection. Thus, wherever possible chip designers provide multiple chip select lines.

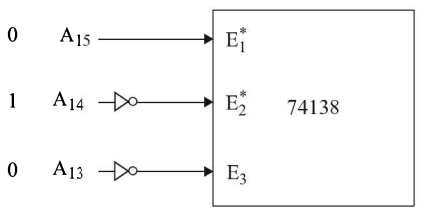


Fig. 11.8
Selection of 74138 when
$A_{15}\,A_{14}\,A_{13}$ = 0 1 0

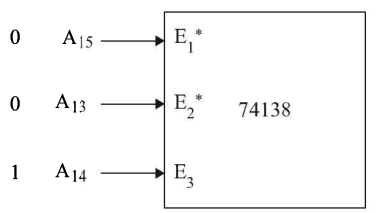


Fig. 11.9
Simpler selection of
74138

■ 11.6 USE OF 74138 IN ALS-SDA-85M KIT

The ALS-SDA-85M kit has a 27128 EPROM of size 16K × 8, and 6116 RAM of size 2K × 8. Also empty sockets are provided in the kit for an expansion EPROM and an expansion RAM. All these four chips are selected using a 74138 IC, as shown in Fig. 11.10.

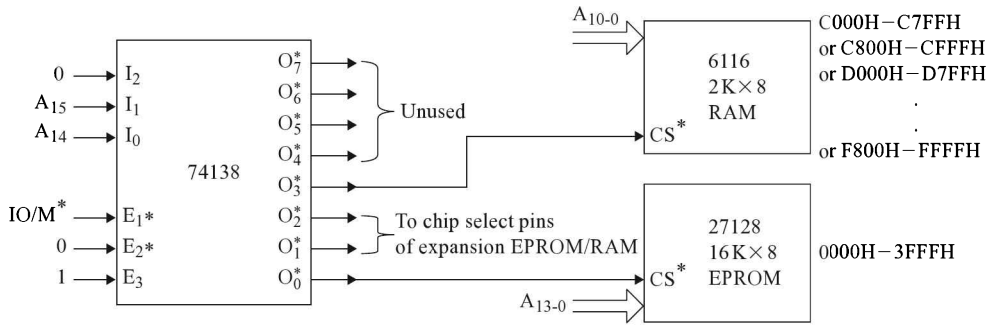


Fig. 11.10 Use of 74138 for selection of memory chips

In this case, the 74138 is selected whenever IO/M* is 0. The 27128 EPROM is selected when A_{15} A_{14} = 0 0. A location within this 16K × 8 EPROM is selected by the remaining address pins $A_{13\text{-}0}$. As such, the lowest and highest address for the 27128 will be as follows.

Lowest address:	0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 0000H
Highest address:	0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 = 3FFFH

The 6116 RAM is selected when A_{15} A_{14} = 1 1. A location within this 2K × 8 RAM is selected by the 11 address pins $A_{10\text{-}0}$. The remaining address pins $A_{13\text{-}11}$ have no role to play, and they are treated as don't cares. As such, the lowest and highest address for the 6116 will be as follows.

Lowest address:	1 1	x x x 0 0 0 0 0 0 0 0 0 0 0
Highest address:	1 1	x x x 1 1 1 1 1 1 1 1 1 1 1

If the don't cares are taken as 0s, the address range will be C000H–C7FFH. If the don't cares are taken as 1s, the address range will be F800H–FFFFH. The other equivalent starting addresses for the RAM are C800H, D000H, D800H, E000H, E800H, and F000H. It was possible to connect a RAM chip with a maximum capacity of 16K × 8 using O_3^* output of 74138. For reasons of economy, only a 2K × 8 RAM chip is used in the ALS kit.

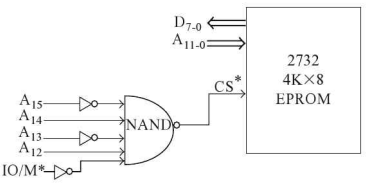
Each of the expansion EPROM/RAM that can be connected using O_1^* and O_2^* outputs of 74138 can have a maximum size of 16K × 8.

There is also one more 74138 IC on the kit, which is used for generating chip select signals for peripheral ICs like 8255, 8253, etc.

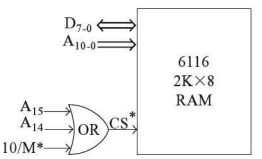
[redacted]

1. Describe the pins that are needed in a typical 8K × 8 RAM chip.
2. What is the need for chip selection in a microcomputer? With a neat diagram explain the interfacing circuit needed to connect a 8K × 8 RAM chip in the address space C000H–DFFFH. Also write another alternative circuit to perform the same function.
3. Is it possible to have a chip select circuit such that 4K × 8 RAM chip has the starting address as C900H? If no, give the reason, and indicate the nearest possible starting address.

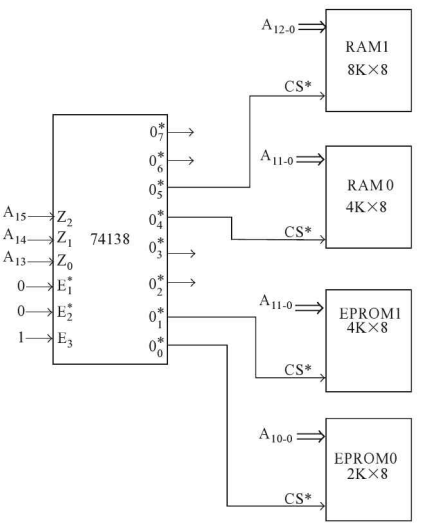
4. For the circuit shown below identify the address range for the EPROM.



5. For the circuit shown below identify the address range for the RAM. In this case, do we have multiple memory address range? If so, identify the different equivalent address ranges.



6. What are the merits and demerits of having more than one chip select pin?
7. What are the merits and demerits of having multiple address ranges for a memory chip?
8. Explain the working of 74138 IC with a functional diagram. What is its application in a microcomputer system?
9. For the circuit shown below identify the address range for the RAMs and the EPROMs. In this case, do you have multiple memory address range for any of the memory chips? If so, identify the different equivalent address ranges.



10. With a neat diagram explain the interfacing circuit using 74138 needed to connect the following.

    a. 2K × 8 RAM chip in the address space C000H–C7FFH,
    b. 4K × 8 RAM chip in the address space D000H–DFFFH,
    c. 2K × 8 EPROM chip in the address space 0000H–07FFH,
    d. 8K × 8 EPROM chip in the address space 2000H–3FFFH.

    In this case, do you have multiple memory address ranges for any of the memory chips? If so, identify the different equivalent address ranges.

12 Addressing of I/O Ports

- Need for I/O ports
  - *Comparison of I/O port chips and memory chips*
- IN and OUT instructions
  - *IN a8 instruction*
  - *OUT a8 instruction*
- Memory-mapped I/O
- I/O-mapped I/O
- Comparison of memory-mapped I/O and I/O-mapped I/O
  - *Merits of I/O-mapped I/O and demerits of memory-mapped I/O*
  - *Demerits of I/O-mapped I/O and merits of memory-mapped I/O*
  - *I/O-mapped I/O or memory-mapped I/O*
- Questions

This chapter reverts to the description of instruction types. Two of the remaining 7 instruction types are explained here. This chapter discusses in detail the importance and need for I/O ports and also describes memory-mapped I/O and I/O-mapped I/O.

12.1 NEED FOR I/O PORTS

CPU and main memory are very fast compared with electromechanical input or output devices like printers, etc. In such a case, it is essential that the data lines of the computer are not kept engaged for a long time during communication with input/output (I/O) devices. Otherwise, the overall speed of the computer system comes down drastically. So I/O devices are connected to a computer through I/O ports.

For example, to communicate with a printer, the CPU loads the output port connected to the printer at electronic speeds. The printer slowly prints this. When the printer has finished printing, the output port requests the CPU for further data. This way, the CPU is allowed to work at its full speed, with no degradation in the overall speed of the computer system.

In fact, to further improve the speed, a printer will have printer buffer. For example, the popular LX-800 Epson dot matrix printer has a printer buffer size of 3 KB. The microcomputer fills this buffer in a fraction of a second. Then as far as the computer is concerned the printing is over! But, the user would not have witnessed the printing of a single character as yet! The printer prints this entire information in about 15–20 s. Then the printer requests for further data to be printed, from the computer.

Thus I/O devices are never directly connected to a computer. Computer and an I/O device always communicate with an I/O port as the middleman.

12.1.1 Comparison of I/O Port Chips and Memory Chips

An I/O port chip also stores information like a memory chip. But, an I/O port chip generally stores only 1 byte of information, and some I/O port chips store a few bytes of information. For example, Intel 8212 I/O port chip stores only 1 byte of information, but Intel 8255 chip can be used to store 3 bytes of information. However, memory chips contain a large number of memory locations like 1K, 4K, 8K, etc. A location within a memory chip is selected by the address pins $A_{n\text{-}0}$.

Second, an I/O port chip will have eight lines for communication with the microprocessor, and another eight lines for communication with the I/O device. An I/O port only acts as a buffer between the microprocessor and the I/O device. But, a memory chip will have only eight lines for communication with the microprocessor.

Third, many I/O port chips are programmable. That is, the user can decide the way the I/O port should work in his system. An example for a programmable I/O port chip is Intel 8255. In this case, the user can program a port as an input port or an output port. Also, the user can program a port to work in one of the several modes like basic I/O, handshake I/O, etc. However, there are I/O port chips, which are non-programmable. An example is Intel 8212. Obviously, programmable I/O ports are more popular than non-programmable ports. But memory chips are only storage chips, and as such are not programmable.

Fig. 12.1a, b, and c indicate the essential pins needed in an output port, an input port, and a memory chip. For simplicity, I/O port chip with a single port is shown.

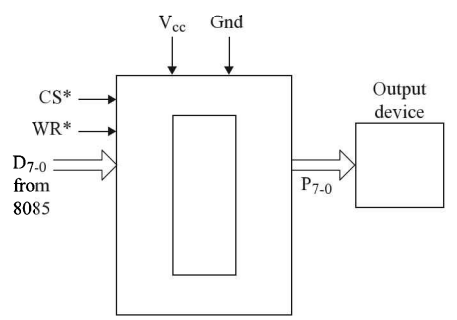


Fig. 12.1a
Pins for an output port chip

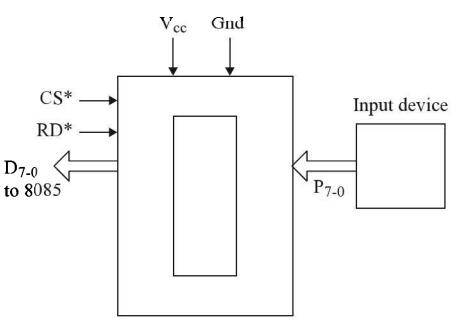


Fig. 12.1b
Pins for an input port chip

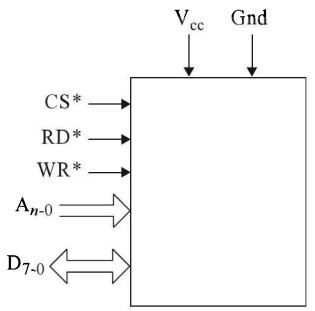


Fig. 12.1c
Pins for a memory chip

■ 12.2 IN AND OUT INSTRUCTIONS

There are two instructions in 8085 for communication with I/O ports. They are the IN and OUT instructions. The IN or OUT instruction mnemonic should be followed by an 8-bit port address. Thus $2^8 = 256$ input ports and 256 output ports are possible in a 8085-based microcomputer.

12.2.1 IN a8 Instruction

IN is a mnemonic that stands for **IN**put to Accumulator from the contents of input port whose 8-bit address is indicated in the instruction as a8. It occupies 2 bytes in memory. The first byte specifies the opcode, and the next byte provides the 8-bit port address. IN EFH is an example instruction of this type. The result of execution of this instruction is shown below with an example.

	Before	***After***
Input port no. EFH	BCH	
Accumulator contents	12H	BCH

IN instruction is the only instruction using which the Accumulator can be loaded with the contents of an input port. A possible chip select circuit to connect an input port with an address as EFH is as shown in Fig. 12.2.

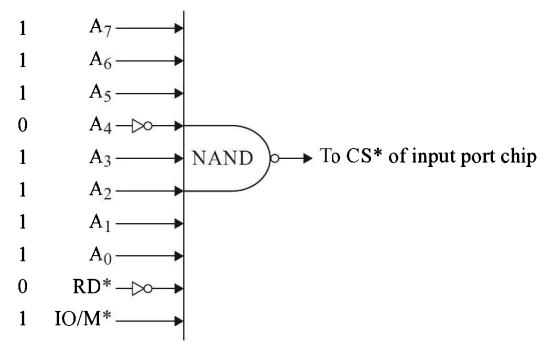


Fig. 12.2
Chip select circuit for input port EFH

Notice that when $A_7\,A_6\,A_5\,A_4\,A_3\,A_2\,A_1\,A_0 = 1\,1\,1\,0\,1\,1\,1\,1$, RD* = 0, and IO/M* = 1, all the inputs to the NAND gate become logic 1, and so the input port chip gets selected. Thus the chip responds when the 8085 sends out address as EFH, IO/M* as 1, and RD* as 0. In other words, we can say that it is input port number EFH.

Summary: IN a8 (2 bytes; IN EFH; 1 opcode)

12.2.2 OUT a8 Instruction

OUT is a mnemonic that stands for **OUT**put Accumulator contents to an output port whose 8-bit address is indicated in the instruction as a8. It occupies 2 bytes in memory. First byte specifies the opcode, and the next byte provides the 8-bit port address. OUT EFH is an example instruction of this type. The result of execution of this instruction is shown below with an example.

	Before	*After*
Accumulator contents	12H	
Output port no. EFH	BCH	12H

OUT instruction is the only instruction using which Accumulator contents can be sent out to an output port. A possible chip select circuit to connect an output port with an address as EFH is as shown in Fig. 12.3.

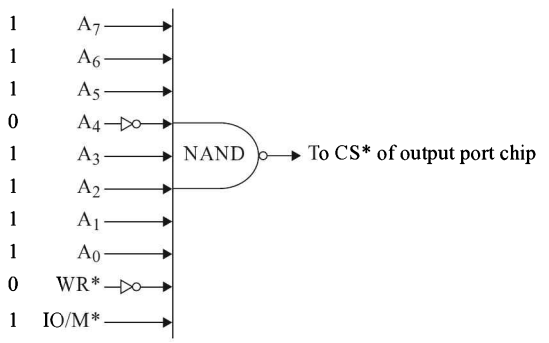


Fig. 12.3
Chip select circuit for output port EFH

Notice that when $A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0$ = 1 1 1 0 1 1 1 1, WR* = 0, and IO/M* = 1, all the inputs to the NAND gate become logic 1, and so the output port chip gets selected. Thus, the chip responds when the 8085 sends out address as EFH, IO/M* as 1, and WR* as 0. In other words, we can say, it is output port number EFH.

Notice that it is possible to have an input port with the address EFH, and an output port with the same address EFH. When the 8085 sends out the address as EFH and IO/M* as 1, only one of them is selected based on the RD* and WR* signals. Thus, it is possible to have a total of 256 input ports and a total of 256 output ports.

Summary: OUT a8 (2 bytes; OUT EFH; 1 opcode)

■ 12.3 MEMORY-MAPPED I/O

It is possible to address an I/O port as if it were a memory location. For example, let us say, the chip select pin of an I/O port chip is activated when address = FFF0H, IO/M* = 0, and RD* = 0. This is shown in Fig. 12.4.

In this case, the I/O port chip is selected when the 8085 is thinking that it is addressing memory location FFF0H for a read operation. Note that 8085 thinks that it is addressing a memory location because it has sent out IO/M* as a logic 0. But in reality, an input port has been selected, and the input port supplies information to the 8085. Such I/O ports that are addressed by the processor as if they were memory locations are called memory-mapped I/O ports.

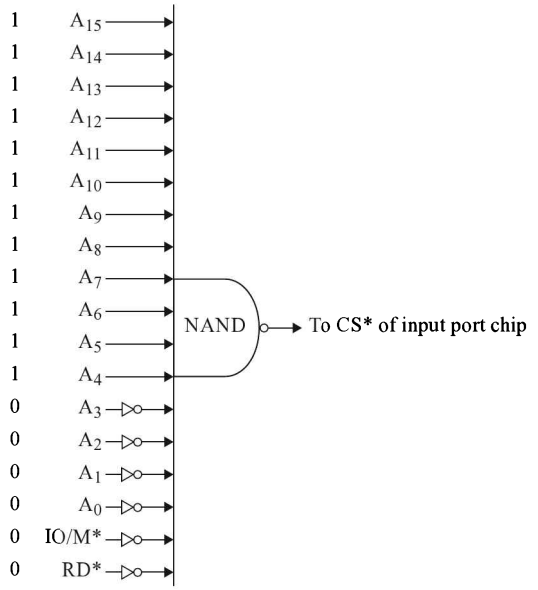


Fig. 12.4
Memory-mapped input port with address FFF0H

■ 12.4 I/O-MAPPED I/O

Generally, a processor like 8085 addresses an I/O port by sending out 8-bit port address and IO/M* = 1. For example, let us say, the chip select pin of an I/O port chip is activated when 8-bit address = F0H, IO/M* = 1, and RD* = 0. This is shown in Fig. 12.5.

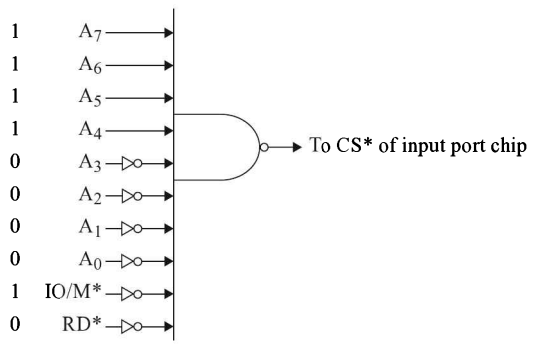


Fig. 12.5
I/O-mapped input port with address F0H

Such I/O ports, which are addressed by the processor by sending out IO/M* as logic 1 are called I/O-mapped I/O ports. Our discussion about I/O ports in Sect. 12.2 of this chapter was confined to only such I/O-mapped I/O ports.

■ 12.5 COMPARISON OF MEMORY-MAPPED I/O AND I/O-MAPPED I/O

In this section the merits and demerits of these two schemes of addressing of I/O ports is discussed.

12.5.1 Merits of I/O-mapped I/O and Demerits of Memory-mapped I/O

1. IN and OUT instructions are used for addressing I/O-mapped I/O ports. The mnemonics of these instructions clearly indicate that the processor communicates with an I/O port. To load accumulator from memory-mapped input port with address FFF0H, we have to execute LDA FFF0H instruction. The mnemonic of this instruction gives the impression that the accumulator is being loaded from memory location FFF0H!
2. Some microprocessors have a control pin to select a memory location or an I/O port. For example, in 8085, IO/M* pin is used to select an I/O port or a memory location. The address sent out on the address pins selects an I/O port or a memory location based on the value sent out on IO/M*. In other words, there are two separate address spaces—one for I/O ports and another for memory, as shown in Fig. 12.6.

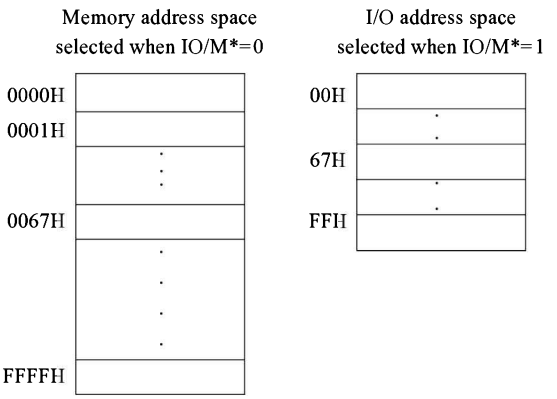


Fig. 12.6
Memory address space and I/O address space for 8085

3. The I/O address space has only 256 locations, as I/O port addresses are only 8 bits in size in 8085. The memory address space has 64K locations, as memory addresses are 16-bits long in 8085. Thus it is possible to have full 64K space exclusively for memory, if we prefer to use I/O-mapped I/O scheme for addressing I/O ports. It is possible to have an I/O-mapped I/O port with the address 67H, and also have a memory location with the address 0067H.
4. If we connect I/O ports as memory-mapped ports with addresses in the range of FF00H–FFF0H, then we should not allot this address range to any memory chip. If memory and a memory-mapped I/O port have the same address, both get selected simultaneously, and this can damage the processor, the memory chip, and the I/O port chip. Thus with memory-mapped I/O, the entire 64K address space (in case of 8085) is not available for memory. This is because some address space is lost to memory-mapped I/O ports.

12.5.2 Demerits of I/O-mapped I/O and Merits of Memory-mapped I/O

1. Only IN and OUT instructions are used for addressing I/O-mapped I/O ports. But a large number of instructions can be used for communication with a memory-mapped I/O port. In fact, all the instructions that can be used to access a memory location can be used to access a

memory-mapped I/O port. Suppose we have a memory-mapped input port with the address FFF0H, then the effect of executing the following instructions is shown alongside in the comments field.

```
LXI H, FFF0H  ;   HL pair loaded with FFF0H
MOV C, M      ;   C register loaded with contents of input port FFF0H
INR M         ;   Increment by 1 contents of input port FFF0H
ANA M         ;   Perform AND of accumulator and input port FFF0H
ADD M         ;   Add accumulator contents and input port FFF0H
```

2. The designer of a microprocessor is required to produce a chip design with minimum possible pins. That is the reason we find multiplexed address and data pins, AD_{7-0}. If I/O-mapped I/O facility is needed, then there must be a pin of the kind IO/M*, to distinguish between I/O addressing and memory addressing. However, if it is decided by the designer of the processor chip that memory and I/O ports will be addressed in the same way, then there is no need of IO/M* pin at all. In Motorola microprocessor chips, there is no IO/M* pin because they use memory-mapped I/O ports. In Motorola 6800 there will be a single address space in which space allocation is made for memory and I/O ports, as shown in Fig. 12.7. The ports are always memory-mapped in Motorola 6800.
3. Using I/O-mapped I/O, only accumulator can communicate with an I/O port. But using memory-mapped I/O any register can communicate with a memory-mapped I/O port. For example, 'MOV C, M' can be used to load C register with contents of a memory-mapped I/O port, whose address is provided in HL pair.
4. Using I/O-mapped I/O, only data transfer operation is possible between accumulator and an I/O port. They are done using the IN and OUT instructions. But using memory mapped I/O, even arithmetic and logical operations can be performed. For example, ADD M can be used to add accumulator and contents of a memory-mapped I/O port, whose address is provided in HL pair. Similarly, ANA M can be used perform AND operation on accumulator and contents of a memory-mapped I/O port, whose address is provided in HL pair.

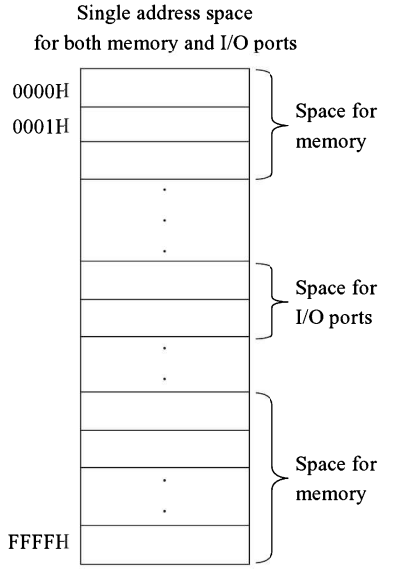


Fig. 12.7
6800 address space for
memory and I/O ports

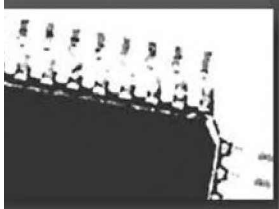
5. Using I/O-mapped I/O, only 256 input ports and 256 output ports could be addressed in a 8085 system, as an I/O port address is only 8-bits wide. But, using memory-mapped I/O, even upto 64K I/O ports can be addressed in a 8085 system, as memory addresses are 16-bits wide. However, this will be at a cost of corresponding reduction in memory addressing capacity.

12.5.3 I/O-MAPPED I/O OR MEMORY-MAPPED I/O?

After the previous discussion, it is not possible to conclude as to which scheme of addressing I/O ports is better. Both have their merits and demerits. Intel family of microprocessors like 8085, 8086, 80386, Pentium, and Zilog family of microprocessors like Z-80, Z-8000, etc. provide I/O-mapped I/O facility, in addition to providing memory-mapped I/O. So some I/O ports can be connected as I/O-mapped I/O ports, and some others as memory-mapped I/O ports in an Intel processor-based system. But Motorola family of microprocessors like 6800, 68000, 68020, etc. provide only memory-mapped I/O. Thus, we can say that an Intel processor is better compared with a Motorola processor, as far as addressing of I/O ports is concerned.

1. Describe the need for I/O ports in a microcomputer system.
2. Compare I/O port chips with memory chips.
3. How many input ports and output ports are possible in 8085-based microcomputer, if the system uses only I/O-mapped I/O?
4. Explain clearly what a memory-mapped I/O port means.
5. With a neat circuit diagram indicate a possible chip select circuit needed to have the address of an I/O-mapped I/O port as E7H.
6. With a neat circuit diagram indicate a possible chip select circuit needed to have the address of a memory-mapped I/O port as 7FE7H.
7. Bring out the merits and demerits of I/O-mapped and Memory-mapped I/O.

13 Architecture of 8085



- Details of 8085 architecture
  - *Arithmetic logic unit (ALU)*
  - *Timing and control unit*
  - *Instruction register (IR)*
  - *W and Z registers*
  - *Temporary (temp) register*
  - *Multiplexer/demultiplexer*
  - *Address/data buffers*
  - *Internal address latch*
  - *Incrementer/decrementer*
  - *Connection of registers to internal bus*
- Instruction cycle
  - *Opcode Fetch (OF) machine cycle*
  - *Memory Read (MR) machine cycle*
  - *Memory Write (MW) machine cycle*
  - *I/O Write (IOW) machine cycle*
  - *I/O Read (IOR) machine cycle*
- Comparison of different machine cycles
- Memory speed requirement
  - *Earliest data output time considering t_{ACC}*
  - *Earliest data output time considering t_{CE}*
  - *Earliest data output time considering t_{OE}*
  - *27128-20 Compatibility check with 8085AH*
  - *Assessing compatibility of 27128-20 with 8085AH-2*
- Wait state generation
- Questions

In the previous chapters the effect of execution of majority of the instructions has already been discussed. Just the programmer's view of 8085 was enough to discuss the effect of executing an instruction. For example, the effect of executing LDA C100H instruction is that accumulator gets loaded with the contents of memory location C100H. But the mechanism by which accumulator gets loaded with contents of memory location C100H was not discussed. To know these details, one has to know the architecture of 8085, which is much more complex than the programmer's view of 8085. To provide an analogy, programmer's view is similar to the driver's view of an automobile. Architectural viewpoint is similar to the mechanic's view of an automobile. Obviously, a mechanic should have more knowledge about the working of an automobile than a driver.

In this chapter, the discussion about 8085 architecture will be confined to the part that helps in an instruction execution. Details of architecture pertaining to interrupt control, serial I/O control, and direct memory access (DMA) control will be discussed in later chapters. This is a very important chapter, and a clear understanding of the architecture of 8085 provides great confidence to design 8085-based systems.

■ 13.1 DETAILS OF 8085 ARCHITECTURE

Shown in Fig. 13.1 is the architecture of 8085. As we are already aware, it has 8-bit ALU, control unit, the general purpose registers A, B, C, D, E, H, and L, and the special purpose SP, PC, and Flags registers.

13.1.1 Arithmetic Logic Unit (ALU)

It basically performs 8-bit arithmetic and logical operations. Accumulator and Temp registers provide the two operands needed in the operations like addition or logical AND operations. The results will be stored in the accumulator. This is done by sending the ALU output to the accumulator via the internal bus. Also, the Flags register is affected based on the result. In an instruction like ADC B, Temp register receives B register value, and Cy flag is also input to the ALU.

In 'DAD B' instruction, it is required to add HL and BC contents, and store the result in HL. This is also performed by the 8-bit ALU, by adding the LS bytes first, and then adding the MS bytes along with any carry generated.

Even increment or decrement of 16-bit registers is done by this ALU. But in Fig. 13.1 incrementer/decrementer unit is separately shown just for convenience.

13.1.2 Timing and Control Unit

This unit is responsible for generating timing signals and control signals. This unit controls all the activities inside the 8085, as well as outside the 8085.

X1, X2, and Clk out pins: To facilitate timing operations in the microcomputer system, there is a clock generator in the control unit of 8085. The complete oscillator circuit, except the quartz crystal is within the chip. Two pins X1 and X2 are brought out of the chip to provide for the external crystal connection. This is shown in Fig. 13.1. Generally a capacitor of about 20pF value has to be connected between X2 and ground to ensure proper starting up of the crystal. There is a divide by 2 counter in the control unit, which divides the crystal frequency by 2. The 8085A can work at an approximate

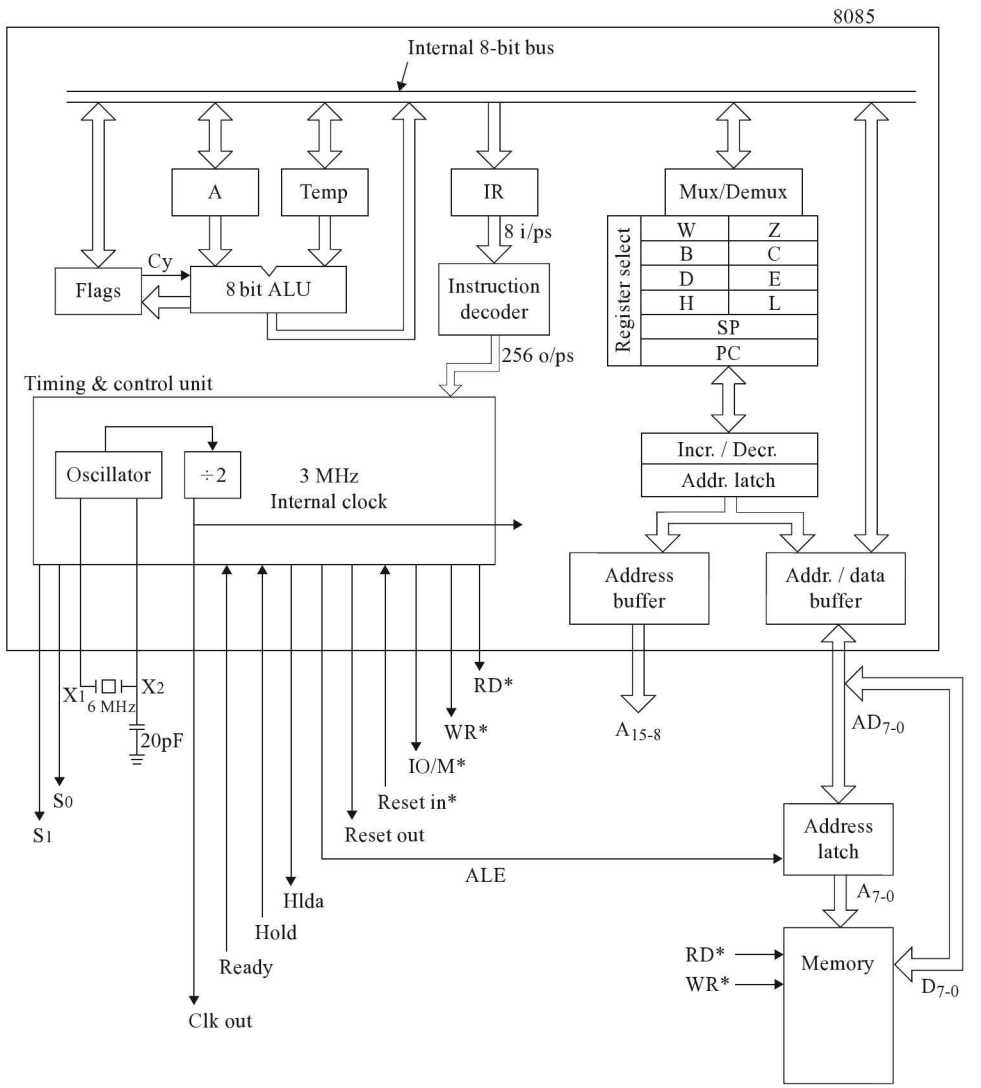


Fig. 13.1 8085 architecture (instruction execution portion only)

maximum clock frequency of 3 MHz internally. Thus typically a 6-MHz crystal is connected between X1 and X2. All operations in a 8085 system take place in synchronization with this clock. This clock signal is brought out on 'clock out' pin of 8085. Peripheral chips, like 8251 USART, which need a clock signal for their operation, use it.

It is also possible to connect a LC-tuned circuit between X1 and X2, instead of a crystal. In very low-cost systems, a resistor can be connected between X1 and X2, and a capacitor connected between X1 and ground to generate the oscillations. However, for good stability of oscillations, it is preferred to use a quartz crystal.

With internal frequency as 3 MHz, the clock period will be 333 nS. In other words, the 8085 clock ticks once every 333 nS, whereas our watch is ticking once every second! A clock cycle of 8085 is also called as a *T* state, *T* standing for 'timing'.

The minimum internal frequency of operation of 8085 (any version) is 500 KHz. In other words, the minimum crystal frequency should be 1 MHz. At lesser frequencies, the information in the registers will be lost. This is because, the registers are basically dynamic RAM cells, which have to be continuously refreshed.

Status signals IO/M, S1 and S0:* An instruction cycle (to be discussed later) requires one to five machine cycles, depending on the instruction. Each machine cycle performs a specific operation, like read from memory. A machine cycle needs a fixed number of clock cycles, the minimum being three clock cycles. The first clock cycle is termed as T1 (T for time period), the second clock cycle as T2, etc.

The type of machine cycle that 8085 is going to execute is indicated by the status signals IO/M*, S1, and S0. This is shown in the following table. These status signals are emitted by the 8085 during T1 of a machine cycle.

*IO/M**	*S1*	*S0*	*Machine cycle*
0	0	1	Memory write (MW)
0	1	0	Memory read (MR)*
0	1	1	Opcode fetch (OF)
1	0	1	I/O write (IOW)
1	1	0	I/O read (IOR)
1	1	1	Interrupt acknowledge (INA)**

*It is bus idle (BI) in the case of DAD instruction.
**It is BI in the case of acknowledge for vectored interrupts.

Generally in a 8085 system, only IO/M* signal is made use of in selecting a memory or an I/O port chip. S1 and S0 signals are not made use of in most systems. For example, in ALS-SDA-85M kit, S1 and S0 signals are unused.

In this chapter, OF, MR, MW, IOR, and IOW machine cycles will be discussed. BI and INA machine cycles will be discussed in the chapter on 8085 interrupts.

Control signals RD, WR*, and INTA*:* During T1, the status signals and address are sent out. Only after the address has become stable, the control signals are emitted by the 8085 during T2 of a machine cycle. The values of the control signals for the various machine cycles are shown in the following table. INTA* signal will be discussed in detail in the chapter on 8085 interrupts.

Machine cycle	*RD**	*WR**	*INTA**
OF, MR, IOR	0	1	1
MW, IOW	1	0	1
INA	1	1	0
BI	1	1	1

During T2 of the halt machine cycle, 8085 tristates RD* and WR*, and emits 1 on INTA.

Address latch enable signal (ALE): As discussed in the previous chapter, 8085 has multiplexed address data pins. So LS byte of address will have to be sent out for only a short time in a machine cycle. During the rest of the machine cycle, the same pins are to be used for data transmission or reception.

Some specialized chips like Intel 8155 have an address latch within the chip, as shown in Fig. 13.2.

During T1 of a machine cycle, 8085 sends out address on $AD_{7\text{-}0}$ and sends out logic 1 on ALE. The address latch in 8155 latches on to the address using this signal. During T2, 8085 stops sending the

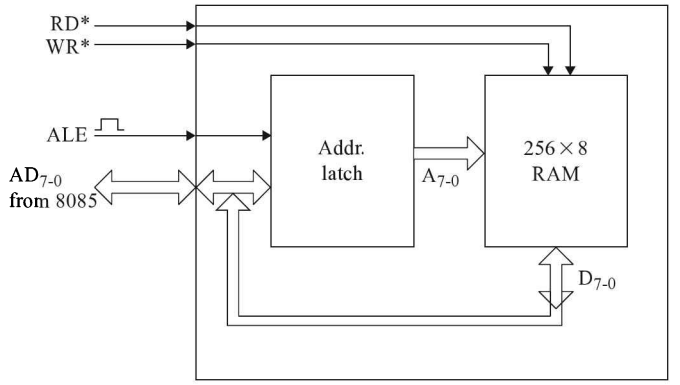


Fig. 13.2
Details of RAM and address latch in Intel 8155

address on AD_{7-0} and sends out logic 0 on ALE. But the latched address is available to the memory portion of 8155 for the complete machine cycle. From T2 onwards, AD_{7-0} is used as D_{7-0} for receiving or transmitting data.

If we are using conventional memory chips, like Intel 6116 RAM chip, they do not have an address latch inside. In such a case, we should have an external address latch as shown in Fig. 13.3.

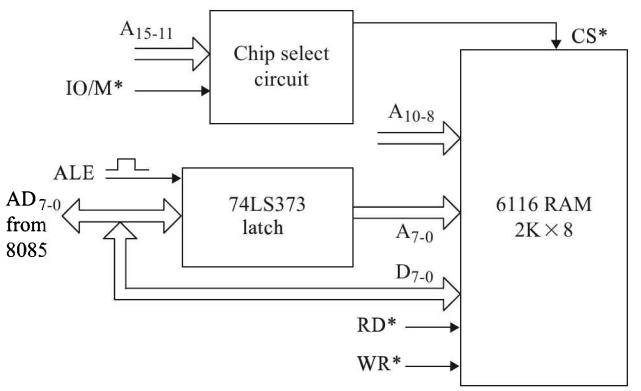


Fig. 13.3
Use of an address latch with a memory chip

During T1 of a machine cycle, 8085 sends out address on AD_{7-0} and sends out logic 1 on ALE. The 74LS373 latches onto the address using this signal. During T2, 8085 stops sending the address on AD_{7-0} and sends out logic 0 on ALE. But the latched address is available to the memory chip for the complete machine cycle. From T2 onwards, AD_{7-0} is used as D_{7-0} for receiving or transmitting data.

Thus ALE signal is used to enable an address latch to perform the latching operation. ALE stands for 'address latch enable', justifying its name. Practical use of 74LS373 is shown in Fig. 13.19, later in the chapter.

Ready input pin: This pin is sensed by the 8085 during T2 of a machine cycle. If it is logic 0, the 8085 understands that the memory or peripheral is not yet ready to send or receive the data. As such, the 8085 will wait the required number of clock cycles for the ready signal to go to logic 1. Only then the 8085 enters T3 state of a machine cycle. Using the Ready signal, it is easy to interface to very slow memory or peripheral chips.

13.1.3 Instruction Register (IR)

This is a 8-bit register. It is used to receive the 8-bit opcode of an instruction from memory. It may be noted that opcode of an instruction is only 8 bits, even though the instruction may be 3-bytes long. IR register is not accessible to the programmer. This means there are no instructions by which a programmer can load a value of his choice into this register. There are no instructions like 'MVI IR, 35H', or 'MOV IR, C'.

Instruction decoder: The instruction decoder receives the input from the IR register. It is an 8-input to 256-output decoder. Depending on the input, only one of the 256 output lines will be activated. The instruction decoder output drives the control unit. The control unit then generates the appropriate control signals to execute the instruction.

13.1.4 W and Z Registers

These are 8-bit registers. They are not accessible to the programmer. They are used for temporarily storing inside the 8085, the 16-bit address operand of an instruction. For example, when 'LDA C234H, instruction is fetched, IR register will receive the opcode for LDA, and W and Z registers will receive C2H and 34H, respectively.

13.1.5 Temporary (Temp) Register

This is an 8-bit register. It is not accessible to the programmer. It temporarily stores inside 8085, the 8-bit operand in an instruction. For example, when 'MVI M, 34H' instruction is fetched, IR register will receive the opcode for MVI M, and Temp register will receive 34H.

In arithmetic and logical operations that involve two operands, the accumulator provides one operand, and the other is provided by the Temp register. For example, in ADD B instruction, B register contents are moved to the Temp register, and then addition of A and Temp register contents is performed by the ALU.

13.1.6 Multiplexer/Demultiplexer

Consider the execution of the instruction 'MOV A, C'. In this case, the 8-bit value in the C register has to be moved to the A register. The registers B, C, D, E, H, and L are connected to the internal bus through a multiplexer/demultiplexer. The register select unit sends the appropriate code to the multiplexer so that C register contents are sent out of the multiplexer to the internal bus. Then Accumulator receives the data from the internal bus.

Similarly, consider the execution of the instruction 'MOV D, A'. In this case, the 8-bit value in the Accumulator has to be moved to the D register. Accumulator sends out the 8-bit value to the internal bus. The registers B, C, D, E, H, and L are connected to the internal bus through a multiplexer/demultiplexer. The register select unit sends the appropriate code to the demultiplexer so that D register receives the contents of the internal bus from the demultiplexer.

13.1.7 Address/Data Buffers

These buffers are bi-directional, when used for data. When used for sending out LS byte of address, they are unidirectional. The buffers are used to increase the current driving capacity. Data comes to

the buffers from the internal bus. LS byte of address comes to the buffers from the internal address latch.

Thus the address/data sent out on AD_{7-0} can drive all the external chips, like RAM chips, EPROM chips, and other peripheral chips. Similarly, the data received by the 8085 from outside is also internally buffered. The received data on AD_{7-0} reaches the internal bus, from where it reaches the final destination.

In fact, in a practical microcomputer, the driving capacity of the data pins, after the internal buffering, may not be adequate. So there will be external buffer chips also. Practical use of such buffer chips is shown in Fig. 13.19, later in the chapter.

Address buffers: These buffers are unidirectional. They are used for sending out the MS byte of address. MS byte of address comes to the buffers from the internal address latch. Thus the address sent out on AD_{15-8} can drive all the external chips, like RAM chips, EPROM chips, and other peripheral chips.

In fact, in a practical microcomputer, the driving capacity of the address pins, after the internal buffering, may not be adequate. So there will be external buffer chips also. Details of such buffer chips are given in the appendix.

13.1.8 Internal Address Latch

The register select unit in the 8085 selects one of the register pairs (BC, DE, HL, SP, PC, or WZ) for sending it to the address latch unit. For example, let us say the contents of PC is C200H. If the register selection unit selects PC, it sends C200H from PC to the internal address latch. The latch holds on to this value, and sends it out on the address pins after buffering. The MS byte of address, C2H, is sent out on A_{15-8}, and the LS byte of address, 00H, is sent out on the pins AD_{7-0}.

A little later, the PC value may be incremented, but still the latch will continue to have the value C200H. This value of C200H is still sent out on the address pins of 8085 even if the PC content is already incremented.

13.1.9 Incrementer/Decrementer

It is actually a function performed by the ALU. But in Fig. 13.1, it is shown as a separate unit just for convenience.

The incrementer is used to increment the PC value, after the 8085 has fetched a byte of the instruction. This way, the PC will be pointing to the next instruction by the time the current instruction is fully fetched. It is also used for incrementing the SP value after a byte is popped out from the stack top. It is also used for incrementing an 8-bit or a 16-bit register.

The decrementer is used for decrementing the SP value before a byte is pushed above the stack top. It is also used for decrementing an 8-bit or a 16-bit register.

13.1.10 Connection of Registers to The Internal bus

Accumulator will have to receive data from internal bus in instructions like 'MOV A, C'. It has to send out data to internal bus in instructions like 'MOV D, A'. As such, the Accumulator communicates with the internal bus in a bi-directional way.

Flags register will have to receive data from internal bus in instructions like POP PSW. It has to send out data to internal bus in instructions like PUSH PSW. As such, the Flags register communicates with the internal bus in a bi-directional way.

In instructions like 'MVI M, 25H', Temp register will have to first of all temporarily receive data value of 25H from internal bus. Later this data in Temp register has to be sent out to memory via the internal bus. As such, the Temp register communicates with the internal bus in a bi-directional way.

IR register will always have to receive the opcode from memory via the internal bus for any instruction. It is never required to send out opcode to the internal bus. As such, the IR register only receives opcode via the internal bus.

Register B will have to receive data from internal bus via demultiplexer in instructions like 'MOV A, B'. It has to send out data to internal bus via multiplexer in instructions like 'MOV B, A'. As such, register B communicates with the internal bus in a bi-directional way. Similarly, registers C, D, E, H, and L also communicate with the internal bus in a bi-directional way.

In instructions like 'LDA 2345H', W and Z registers will have to first of all temporarily receive address value of 2345H from the internal bus. Later they have to send out this address in W and Z registers to memory via the internal bus. As such, W and Z registers communicate with the internal bus in a bi-directional way.

■ 13.2 INSTRUCTION CYCLE

Program and data are stored in memory, which is external to the microprocessor. To execute an instruction of the program, the following steps are to be performed by the 8085.

1. Fetch the opcode from memory;
2. Decode the opcode to identify the instruction;
3. Fetch the remaining bytes, if instruction length is 2 or 3 bytes;
4. Execute the instruction.

These steps put together constitute the instruction cycle. These steps are described in detail with the help of Fig. 13.1 taking several instructions as examples. These instructions are assumed to be in memory, at memory locations as shown in the following.

Memory address	*Memory contents*	*Instruction*
C000H	78H	MOV A, B
C001H	4EH	MOV C, M
C002H	36H	MVI M, 25H
C003H	25H	
C004H	32H	STA D525H
C005H	25H	
C006H	D5H	
C007H	F5H	PUSH PSW
C008H	D3H	OUT 25H
C009H	25H	
C00AH	DBH	IN 35H
C00BH	35H	

Example 1

Execution of 'MOV A, B' instruction. Let us say that PC has C000H as its contents. Then it means that 8085 desires to fetch and execute the instruction starting at memory location C000H. Let us say, at location C000H, we have 78H, which is the opcode for the instruction 'MOV A, B'. This instruction

needs only four clock cycles for the opcode fetch and execution. The action performed by the 8085 in these four clock cycles is explained in the following.

The content of PC is sent to address latch inside the 8085. It latches on to the value C000H. This value of C000H is sent out on the address pins of 8085 after buffering. The MS byte of address, C0H, is sent out on A_{15-8}, and the LS byte of address, 00H, is sent out on the pins AD_{7-0}. This is done during the beginning of T1.

In addition to address information, status signals as shown here are activated by the control unit during T1.

ALE = 1, indicating address present on AD_{7-0}
IO/M* = 0, indicating that the address is meant for memory
S1 = 1, S0 = 1 indicating that it is opcode fetch machine cycle

Address information is sent out on AD_{7-0} only during T1. In T2, the 8085 stops sending out the address and tristates AD_{7-0}. It expects that memory chip should now drive the AD_{7-0} pins with the opcode.

Logic 1 value is sent out on ALE pin during T1. During this clock period, LS byte of address is sent out on AD_{7-0}. The address latch does latching of this LS byte of address, when ALE is made 0 before the end of T1. Only during the beginning of T2, RD* will be made 0. This enables the memory to output the opcode after the access time of the memory chip.

At the end of T2, the 8085 will sense Ready input. If it is 0, the 8085 understands that memory is not yet ready to supply the opcode. In such a case, the next clock cycle is not termed as T3. It will be termed as T_{wait}. At the end of T_{wait} Ready input will be sensed once again. If it is still logic 0, it enters another T_{wait} state once again. The 8085 goes to T3 state only when Ready signal is at logic 1 when sensed by the 8085 at the end of T2 or T_{wait}.

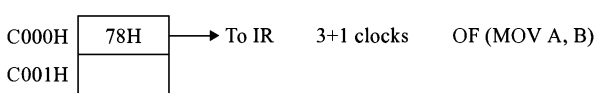
It is the responsibility of the system designer to introduce appropriate number of wait states if the memory or peripherals are slow. However, present-day memory chips and peripherals are so fast that wait states will not be needed for 8085 working at 3 MHz. In such a case, the ready pin can be permanently tied to logic 1. In the ALS kit, ready pin is tied to logic 1. Memory access time requirement calculations are shown later in the chapter.

Thus at the beginning of T3 the opcode will be received by the 8085 in the IR register on AD_{7-0} pins. During the middle of T3, RD* is deactivated by the 8085. This results in floating of AD_{7-0} lines. Also, Ready input is deactivated.

Thus in three clock cycles (if there are no wait states), the opcode is received in IR register. The decoding is done in T4 state. The control unit understands that the opcode corresponds to the 1-byte instruction whose mnemonic is 'MOV A, B'. Then it generates control signals such that B register contents are moved to A the register, via the multiplexer and the internal bus. This decoding and execution is completed in T4.

In fact, after activating RD* during the beginning of T2, the 8085 increments the PC value using the incrementer logic. Thus PC value would have become C001H by the time the opcode is received in IR. That is why it is said that PC points to the instruction to be executed next.

Fig. 13.4 depicts opcode fetch, decode, and execution of MOV A, B instruction in a simple way.



Total of 4 clock cycles with 1 machine cycle

Fig. 13.4
Illustrating MOV A, B instruction execution

13.2.1 Opcode Fetch (OF) Machine Cycle

These four clock cycles in Fig. 13.4 constitute the OF machine cycle. But in these four clock cycles opcode fetch, decode, and also execution were completed. However, in 2- and 3-byte instructions, and also in some 1-byte instructions like 'MOV B, M', only OF and decode operations will be completed in these four clock cycles. Thus OF machine cycle consists of opcode fetch and decode operation, and in some cases execution also. For some instructions like DCX B, the OF machine cycle uses six states.

Henceforth, we just indicate for a machine cycle, like OF, the number of T states needed when there are no wait states. The wait states, if needed, depends on the speed of the memory and peripheral chips used in the system. Waveforms for OF machine cycle, which uses four T states are shown in Figs. 13.5 and 13.6.

Example 2

Execution of 'MOV C, M' instruction. Let us say, at location C001H, we have 4EH, which is the opcode for the instruction MOV C, M. This instruction needs seven clock cycles for the OF, decode, and execution. The first four clock cycles constitute the OF machine cycle, and the next three clock cycles will be memory read (MR) machine cycle. The action performed by the 8085 in these seven clock cycles is the topic of following explanation.

In the first three clock cycles 4EH will be received in IR register from memory location C001H. The decoding is done in T4 state. The control unit understands that the opcode corresponds to the 1-byte instruction whose mnemonic is MOV C, M. These four T states constitute the OF machine cycle, described earlier. In the meanwhile, PC contents would have been incremented to C002H. But C002H will not be sent out from PC at this point, as the instruction MOV C, M is not yet executed.

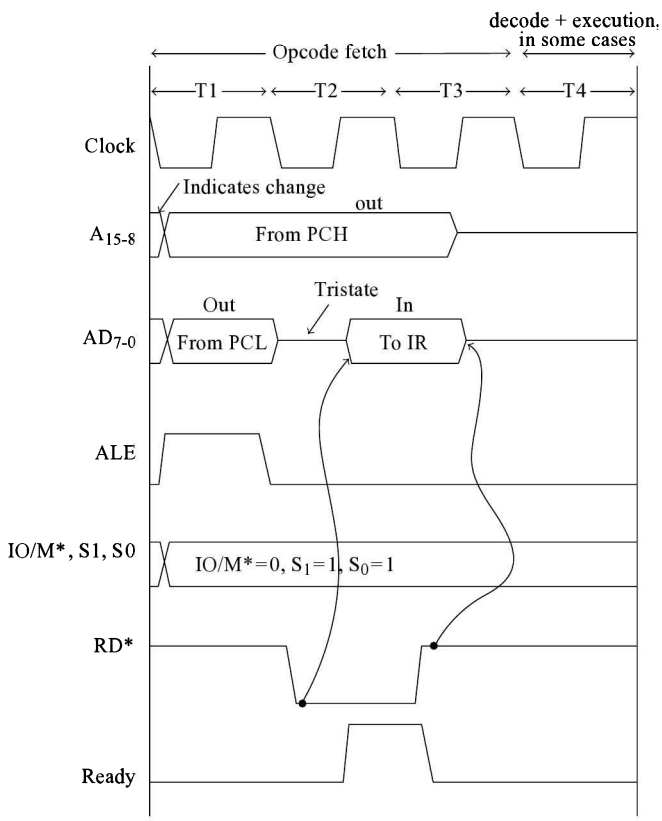


Fig. 13.5
Waveforms for OF machine cycle (without wait state)

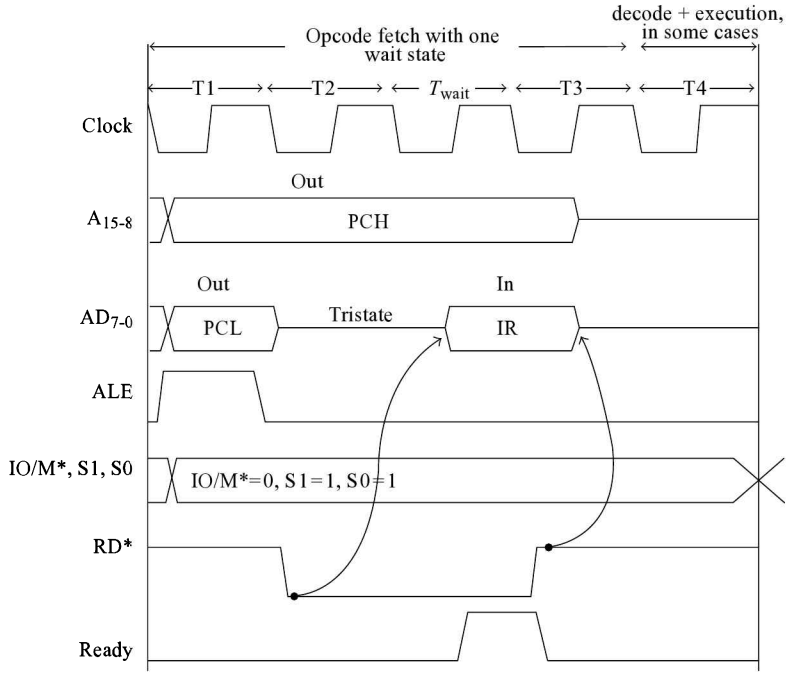


Fig. 13.6 Waveforms for OF machine cycle (with one wait state)

The control unit understands that for instruction execution, information should be received in C register from memory location whose address is present in HL pair. This is performed in a MR machine cycle using three clock cycles, as described in the following.

Let us say HL pair has the value D300H and the content of memory location D300H is 45H. Then 45H should be received in the C register. The control unit directs the register select unit to select HL pair. Then HL contents are latched by the internal address latch, and then buffered and sent out on A_{15-8} and AD_{7-0} pins. This is done during T1 of MR machine cycle.

In addition to address information, status signals as shown in the following are activated by the control unit during T1.

ALE = 1, indicating address present on AD_{7-0};
IO/M* = 0, indicating that the address is meant for memory;
S1 = 1, S0 = 0 indicating that it is MR machine cycle.

Address information is sent out on AD_{7-0} only during T1. In T2, the 8085 stops sending out the address and tristates AD_{7-0}. It expects that memory chip should now drive the AD_{7-0} pins with the data.

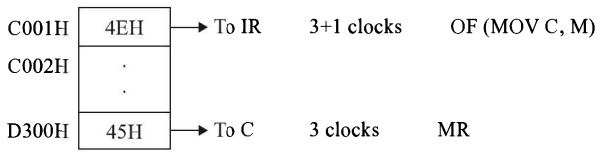
Logic 1 value is sent out on ALE pin during T1. During this clock period, LS byte of address is sent out on AD_{7-0}. The address latch does latching of this LS byte of address, when ALE is made 0 before the end of T1. Only during the beginning of T2, RD* will be made 0. This enables the memory to output the data after the access time of the memory chip.

At the end of T2, the 8085 will sense Ready input. If it is 0, the 8085 understands that memory is not yet ready to supply the data. In such a case, the next clock cycle will be T_{wait}. During the end of T_{wait}, Ready input will be sensed once again. If it is still logic 0, once again it enters another T_{wait} state. The 8085 goes to T3 state only when ready signal is at logic 1 when sensed by the 8085 at the end of T2 or T_{wait}.

Thus at the beginning of T3 the data will be received by the 8085 in the C register, on AD_{7-0} pins. During the middle of T3, RD* is deactivated by the 8085. This results in floating of AD_{7-0} lines. Thus in three clock cycles (if there are no wait states), the data is received in C register via the internal bus and the demultiplexer.

In this case, after activating RD* during the beginning of T2, the 8085 does not increment the PC value. Thus PC value remains as C002H when the data is received in C register. If PC were to be incremented, the 8085 had to wrongly skip the instruction at memory location C002H! It can be verified that PC value will have to be incremented in a MR machine cycle only when PC value is sent out as address.

Fig. 13.7 depicts OF, decode, and execution of 'MOV C, M' instruction in a simple way. Thus 'MOV C, M' instruction needs two machine cyles. The first machine cycle, denoted as M1, is of type OF with four T states, and the second machine cycle, denoted as M2, is of type MR with three T states.

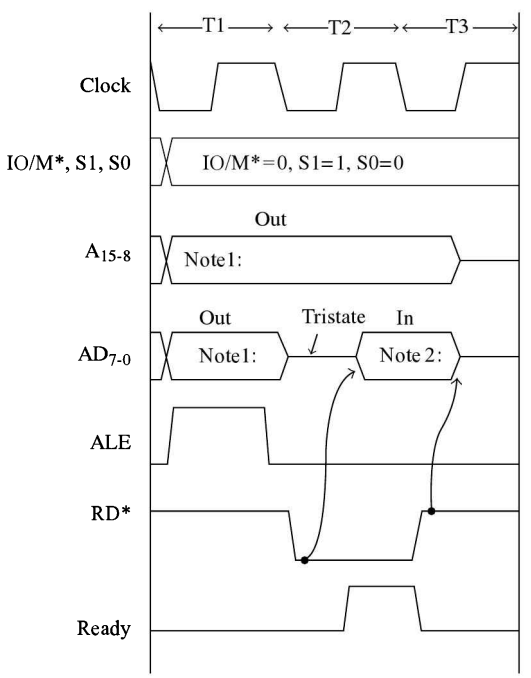


Total of 7 clock cycles with 2 machine cycles

Fig. 13.7
Illustrating MOV C, M instruction execution

13.2.2 Memory Read (MR) Machine Cycle

The last three clock cycles in 'MOV C, M' instruction is an example for MR machine cycle. Waveforms for MR machine cycle is shown in Fig. 13.8.



Note1: Addresses sent out from PC, BC, DE, HL, SP or WZ
Note2: Data received in any 8-bit register except IR

Fig. 13.8
Waveforms for MR machine cycle (without wait state)

The address sent out from a register pair in a MR machine cycle depends on the MR machine cycle within an instruction as shown in the following table.

Instruction	*Operation*	*Address reg.*
LDA 1234H	Loading W with 12H (or Z with 34H)	PC
POP B	Popping information from stack top	SP
MOV C, M	Loading C from memory pointed by HL	HL
LDAX B	Loading A from memory pointed by BC	BC
LDAX D	Loading A from memory pointed by DE	DE
LDA 1234H	Loading A from memory location 1234H	WZ

The data received in a register during a MR machine cycle depends on the MR machine cycle within an instruction as shown in the following. Here, 'r' stands for any of the registers A, B, C, D, E, H, or L.

Instruction	*Operation*	*Data reg.*
MOV r, M	Loading r from memory pointed by HL	r
POP PSW	Loading flags from stack top	Flags
MVI M, 25H	Loading Temp with 25H from memory	Temp

Example 3

Execution of 'MVI M, 25H' instruction. Let us say, at location C002H, we have 36H, which is the opcode for the instruction 'MVI M, d8'. This instruction needs ten clock cycles for the OF, decode, and execution. The first four clock cycles constitute the OF machine cycle, the next three clock cycles will be MR machine cycle, and the last three clock cycles will be MW machine cycle. The action performed by the 8085 in these ten clock cycles is the topic of following explanation.

In the first three clock cycles 36H will be received in IR register from memory location C002H. The decoding is done in T4 state. The control unit understands that the opcode corresponds to the 2-byte instruction whose mnemonic is MVI M. These four *T* states constitute the OF machine cycle, described earlier. In the meanwhile, PC contents would have been incremented to C003H.

As the control unit understands that MVI M instruction is 2-bytes long, it decides to fetch from the next memory location the second byte of the instruction. The 8085 performs a MR machine cycle, by sending out the PC value of C003H, and receiving 25H in the Temp register. This MR operation takes three clocks as described earlier. In the meanwhile, PC contents would have been incremented to C004H.

Only now the control unit has understood that the instruction to be executed is 'MVI M, 25H'. The control unit understands that for instruction execution, 25H in Temp register should be moved to memory location whose address is present in HL pair. This is performed in a MW machine cycle using three clock cycles, whose description follows.

Let us say HL pair has the value D300H. Then contents of D300H should change to 25H. The control unit directs the register select unit to select HL pair. Then HL contents are latched by the internal address latch, and then buffered and sent out on A_{15-8} and AD_{7-0} pins. This is done during T1 of MW machine cycle.

In addition to address information, status signals as shown in the following are activated by the control unit during T1.

ALE = 1, indicating address present on AD_{7-0};
IO/M* = 0, indicating that the address is meant for memory;
S1 = 0, S0 = 1, indicating that it is MW machine cycle.

Address information is sent out on AD_{7-0} only during T1. At the beginning of T2, the 8085 stops sending out the address on AD_{7-0} and starts sending out the data value of 25H from the Temp register. This is because, in a MW machine cycle, the 8085 only has to send out the data.

Logic 1 value is sent out on ALE pin during T1. During this clock period, LS byte of address is sent out on AD_{7-0}. The address latch does latching of this LS byte of address, when ALE is made 0 before the end of T1. Only during the beginning of T2, WR* will be made 0. This enables the memory to be written with the data after the access time of the memory chip.

At the end of T2, the 8085 will sense Ready input. If it is 0, the 8085 understands that memory has not yet received the data. In such a case, the next clock cycle will be T_{wait}. At the end of T_{wait}, Ready input will be sensed once again. If it is still logic 0, it enters another T_{wait} state once again. The 8085 goes to T3 state only when Ready signal is at logic 1 when sensed by the 8085 at the end of T2 or T_{wait}.

Thus, by the beginning of T3 the data on AD_{7-0} would have been definitely written to the selected memory location. During the middle of T3, WR* is deactivated by the 8085. This results in disabling the selected memory chip, and terminating the write operation. Thus, in three clock cycles (if there are no wait states), the data is written to selected memory location via the Temp register, the internal bus, and the address/data buffer.

In this case, after activating WR* during the beginning of T2, the 8085 does not increment the PC value. Thus, PC value remains as C004H when memory location D300H is written with the data value of 25H. If PC were to be incremented, the 8085 would wrongly skip the instruction at memory location C004H! It can be verified that PC value should never be incremented in a MW machine cycle.

Fig. 13.9 depicts OF, decode, and execution of 'MVI M, 25H' instruction in a simple way. In this figure, the original content of D300H is shown as 52H. Thus 'MVI M, 25H' instruction needs three machine cycles. M1 is of type OF with four T states, M2 is of type MR with three T states, and M3 is of MW type with three T states, for a total of ten T states.

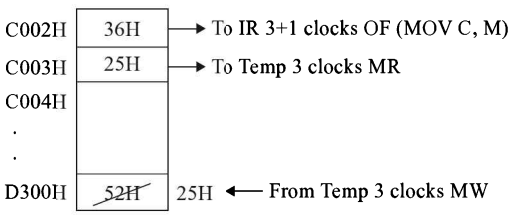


Fig. 13.9
Illustrating 'MVIM,25H' instruction execution

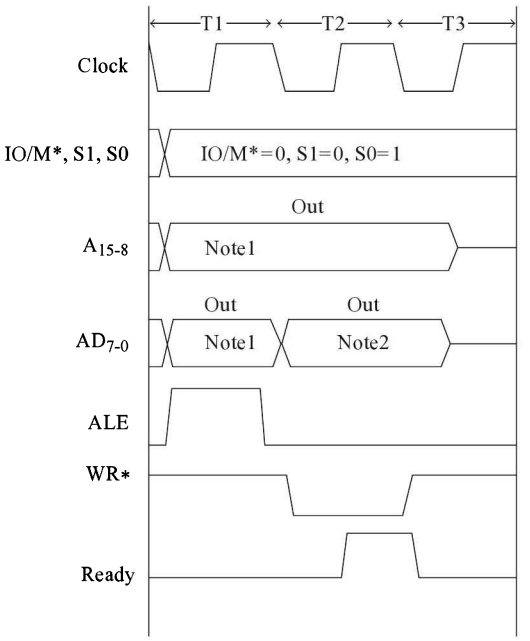
Total of 10 clock cycles with 3 machine cycles

13.2.3 Memory Write (MW) Machine Cycle

The last three clock cycles in 'MVI M, 25H' instruction is an example for MW machine cycle. Waveforms for MW machine cycle are shown in Fig. 13.10.

The address sent out from a register pair in a MW machine cycle depends on the MW machine cycle under consideration as shown in the following.

Reg. pair	*Example*
SP	Pushing information above stack top in PUSH B
HL	Saving C register in memory pointed by HL in MOV M, C
BC	Saving A register in memory pointed by BC in STAX B
DE	Saving A register in memory pointed by DE in STAX D
WZ	Saving A register in location 1234H in STA 1234H



Note1: Addresses sent out from BC, DE, HL, SP or WZ (but never from PC)
Note2: Data sent out from any 8-bit register except IR, W and Z

Fig. 13.10
Waveforms for MW machine cycle (without wait state)

The data sent out from a register during a MW machine cycle depends on the MW machine cycle within an instruction as shown below. Here, 'r' stands for any of the registers A, B, C, D, E, H, or L.

r	Saving register r in memory pointed by HL in 'MOV M, r'
Flags	Pushing information above stack top in 'PUSH PSW'
Temp	Saving of 25H in memory pointed by HL in 'MVI M, 25H'

The address sent out from a register pair in a MW machine cycle depends on the MW machine cycle within an instruction as shown in the following.

Instruction	*Operation*	*Address reg.*
PUSH B	Push information above stack top	SP
MOV M, C	Save C reg. in memory pointed by HL	HL
STAX B	Save A reg. in memory pointed by BC	BC
STAX D	Save A reg. in memory pointed by DE	DE
STA 1234H	Save A reg. in location 1234H	WZ

The data sent out from a register during a MW machine cycle depends on the MW machine cycle within an instruction as shown in the following. Here, 'r' stands for any of the registers A, B, C, D, E, H, or L.

Instruction	*Operation*	*Data reg.*
MOV M, r	Save register r in memory pointed by HL	r
PUSH PSW	Push flags information above stack top	Flags
MVI M, 25H	Save 25H in memory pointed by HL	Temp

Example 4

Execution of STA D525H instruction. Let us say, at location C004H, we have 32H, which is the opcode for the instruction 'STA a16'. This instruction needs 13 clock cycles for the OF, decode, and execution. It uses an OF machine cycle of four clocks, followed by two MR and a MW machine cycle, each of three clocks. The action performed by the 8085 in these 13 clock cycles is explained in the following discussion.

In the first three clock cycles, 32H will be received in IR register from memory location C004H. The decoding is done in T4 state. The control unit understands that the opcode corresponds to the 3-byte instruction whose mnemonic is STA. These four *T* states constitute the OF machine cycle, described earlier. In the meanwhile, PC contents would have been incremented to C005H.

As the control unit understands that STA instruction is 3-bytes long, it decides to fetch from the next two memory locations the second and third bytes of the instruction. The 8085 performs a MR machine cycle by sending out the PC value of C005H, and by receiving 25H in the Z register. This MR operation takes three clocks as described earlier. In the meanwhile, PC contents would have been incremented to C006H.

The 8085 then performs another MR machine cycle by sending out the PC value of C006H, and by receiving D5H in the W register. This MR operation takes another three clocks. In the meanwhile, PC contents would have been incremented to C007H.

Only now the control unit has understood that the instruction to be executed is STA D525H. The control unit understands that for instruction execution, contents of the accumulator should be stored in memory location whose address is present in WZ pair. This is performed in a MW machine cycle using three clock cycles. If the Accumulator has the value 25H, then contents of D525H will change to 25H. As this is a MW machine cycle, PC value remains unaltered at C007H.

Fig. 13.11 depicts OF, decode, and execution of STA D525H instruction in a simple way. In this figure, the original content of D525H is shown as 52H.

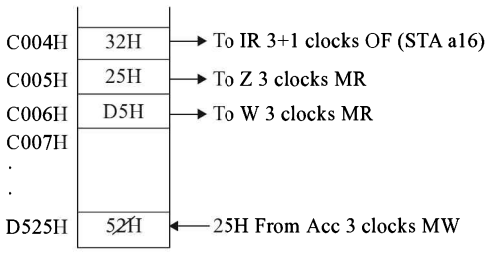


Fig. 13.11
Illustrating 'STA D525H' instruction execution

Example 5

Execution of PUSH PSW instruction. Let us say, at location C007H, we have F5H, which is the opcode for the instruction PUSH PSW. This instruction needs 12 clock cycles for the OF, decode, and execution. It uses an OF machine cycle of six clocks, followed by two MW machine cycles, each of three clocks. The action performed by the 8085 in these 12 clock cycles is explained in the following.

In the first three clock cycles, F5H will be received in the IR register from memory location C007H. The decoding is done in T4 state. The control unit understands that the opcode corresponds to the single-byte instruction whose mnemonic is PUSH PSW. In the meanwhile, PC contents would have been incremented to C008H. In the next two clock cycles of this OF machine cycle, SP value will be decremented by 1. These six T states constitute the OF machine cycle in this case.

By the end of the T4 state in the OF machine cycle, 8085 understands that for instruction execution, contents of the Accumulator and the Flags register should be stored above the stack top. Suppose

SP content is C708H, Accumulator content is 30H, and Flags value is 40H, then C707H contents should change to 30H, and C706H contents should change to 40H.

To push accumulator above the stack top, first of all SP contents have to be decremented from C708H to C707H using the decrementer. This is performed in T5 and T6 states of OF machine cycle. Only then MW machine cycle begins by sending out the address C707H from SP, and the data 30H from the Accumulator. As it is a MW machine cycle, PC value remains unaltered at C008H. However, in the meanwhile the SP value would be decremented from C707H to C706H. This MW machine cycle uses three clock cycles.

Then the next MW machine cycle begins by sending out the address C706H from SP, and the data 40H from Flags register. As it is a MW machine cycle, PC value remains unaltered at C008H. Even SP value remains unaltered at C706H in this case. This MW machine cycle uses three clock cycles.

Fig. 13.12 depicts OF, decode, and execution of PUSH PSW instruction in a simple way. In this figure, the original contents of C706H and C707H are shown as 52H and 42H, respectively.

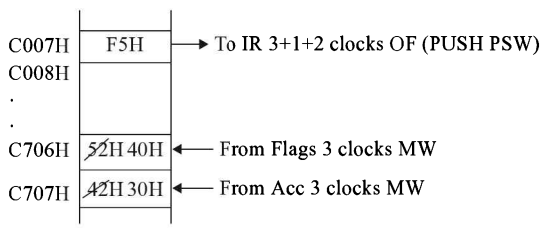


Fig. 13.12
Illustrating 'PUSH PSW' instruction execution

The OF machine cycle uses six *T* states in the following cases.

1. When the instruction execution requires storing information above the stack top. Instructions in this category are PUSH rp, CALL a16, RST*n*, and conditional call instructions.
2. When the instruction is INX rp or DCX rp.
3. When the instruction is PCHL or SPHL.
4. When it is a conditional return instruction.

Example 6

Execution of OUT 25H instruction. Let us say, at location C008H, we have D3H, which is the opcode for the instruction 'OUT a8'. This instruction needs ten clock cycles for the OF, decode, and execution. It uses an OF machine cycle of four clocks, followed by a MR and an IOW machine cycle, each of three clocks. The action performed by the 8085 in these ten clock cycles is explained in the following.

In the first three clock cycles D3H will be received in IR register from memory location C008H. The decoding is done in T4 state. The control unit understands that the opcode corresponds to the 2-byte instruction whose mnemonic is OUT. These four T states constitute the OF machine cycle. In the meanwhile, PC contents would have been incremented to C009H.

As the control unit understands that the OUT instruction is 2-bytes long, it decides to fetch the second byte of the instruction from the next memory location. The 8085 performs a MR machine cycle, by sending out the PC value of C009H, and by receiving 25H in the W register and also the Z register. The point to be noted here is that both W and Z registers are loaded with the same value, 25H. This MR operation takes three clocks. In the meanwhile, PC contents would have been incremented to C00AH.

Only now the control unit has understood that the instruction to be executed is OUT 25H. The control unit understands that for instruction execution, contents of the Accumulator should be stored in an output port whose address is 25H. This is performed in an I/O write (IOW) machine cycle using three clock cycles.

If the Accumulator has the value D5H, then contents of the output port 25H will change to D5H. Just as with a MW machine cycle, PC value remains unaltered at C00AH in this IOW machine cycle. Fig. 13.13 depicts OF, decode, and execution of OUT 25H instruction in a simple way. In this figure, the original content of output port 25H is shown as 52H.

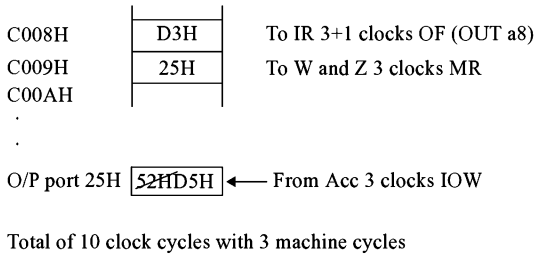


Fig. 13.13
Illustrating 'OUT 25H' instruction execution

13.2.4 I/O Write (IOW) Machine Cycle

The last three clock cycles in OUT 25H instruction is an example for IOW machine cycle. Waveforms for IOW machine cycle are shown in Fig. 13.14.

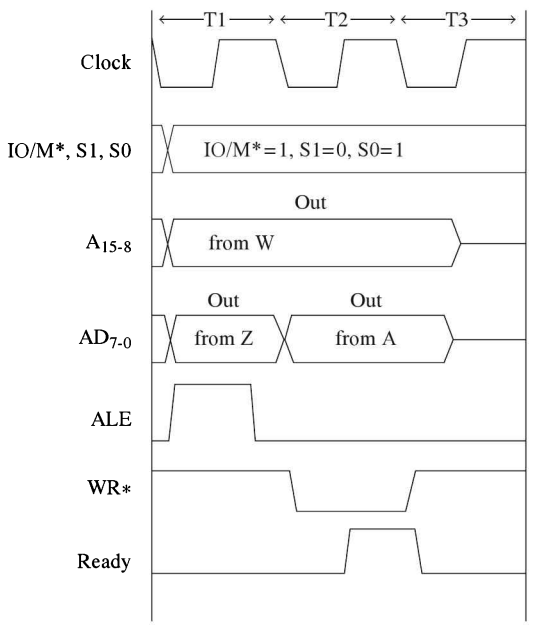


Fig. 13.14
Waveforms for IOW machine cycle (without wait state)

Note that in an IOW machine cycle, W and Z will have identical 8-bit port address. There is a definite advantage because of address duplication on A_{15-8} and AD_{7-0} when using 8755 (2K × 8 EPROM and two 8-bit ports) and 8155 (combination of 256 × 8 RAM, 3 I/O ports, and 14-bit timer). A 8085-based

microcomputer can be formed using only these two chips in addition to 8085. However, these chips are not in favour these days, except when minimum chip microcomputer configuration is desired.

The 16-bit value in register pair WZ will be sent out as the address in the IOW machine cycle. Also, note that in an IOW machine cycle, only the Accumulator contents will be sent out as the data to the addressed output port. In 8085, we come across IOW machine cycle only during the execution of OUT a8 instruction.

Example 7

Execution of IN 35H instruction. Let us say, at location C00AH, we have DBH, which is the opcode for the instruction 'IN a8'. This instruction needs ten clock cycles for the OF, decode, and execution. It uses an OF machine cycle of four clocks, followed by a MR and an IOR machine cycle, each of three clocks. The action performed by the 8085 in these ten clock cycles is explained in the following.

In the first three clock cycles, DBH will be received in IR register from memory location C00AH. The decoding is done in the T4 state. The control unit understands that the opcode corresponds to the 2-byte instruction whose mnemonic is IN. These four *T* states constitute the OF machine cycle. In the meanwhile PC contents would have been incremented to C00BH.

As the control unit understands that IN instruction is 2-bytes long, it decides to fetch the second byte of the instruction from the next memory location. The 8085 performs a MR machine cycle by sending out the PC value of C00BH, and by receiving 35H in the W register and also the Z register. The point to be noted here is that both W and Z registers are loaded with the same value 35H. The advantage of address duplication in WZ pair in IN a8 instruction is same as that in a OUT a8 instruction. This MR operation takes three clocks. In the meanwhile, PC contents would have been incremented to C00CH.

Only now the control unit has understood that the instruction to be executed is IN 35H. The control unit understands that for instruction execution, the Accumulator should be loaded with contents of the input port whose address is 35H.

This is performed in a I/O read (IOR) machine cycle using three clock cycles. If input port 35H has the value D5H, then contents of the Accumulator will change to D5H. Just as with a IOW machine cycle, PC value remains unaltered at C00CH in this IOR machine cycle.

Fig. 13.15 depicts OF, decode, and execution of IN 25H instruction in a simple way. In this figure, the original content of the Accumulator is shown as 52H.

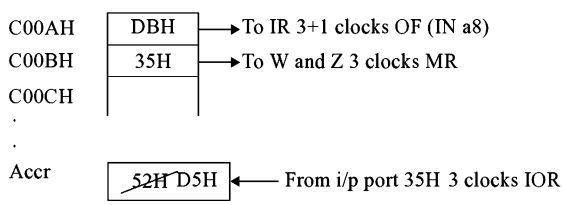


Fig. 13.15
Illustrating 'IN 35H' instruction execution

13.2.5 I/O Read (IOR) Machine Cycle

The last three clock cycles in IN 35H instruction is an example for IOR machine cycle. Waveforms for IOR machine cycle are shown in Fig. 13.16.

Note that in an IOR machine cycle, W and Z will have identical 8-bit port address. The 16-bit value in register pair WZ will be sent out as the address in an IOR machine cycle. Also, note that in an IOR machine cycle, only the Accumulator receives the data from the addressed input port. In 8085, we come across IOR machine cycle only during the execution of IN a8 instruction.

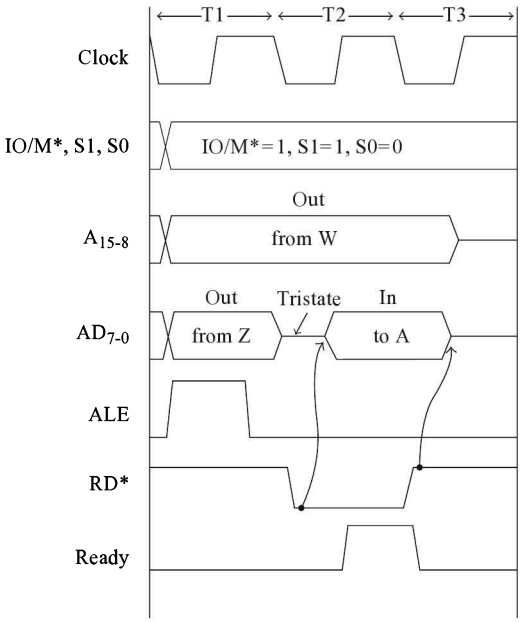


Fig. 13.16
Waveforms for IOR machine cycle (without wait state)

■ 13.3 COMPARISON OF DIFFERENT MACHINE CYCLES

So far we have come across OF, MR, MW, IOR, and IOW machine cycles. The other possible machine cycles in 8085 are BI (bus idle) and INA (interrupt acknowledge) machine cycles. They will be discussed in the chapter on 8085 interrupts. Now the differences between some of the machine cycles are presented in the following tables.

Differences between OF and MR

	OF	*MR*
■ No. of T states	4	3
■ Status of S1 and S0	S1 = 1, S0 = 1	S1 = 1, S0 = 0
■ Data received in	IR only	any byte register except IR*
■ Address sent out from	PC	PC, BC, DE, HL, SP, or WZ
■ PC incremented by	1	by 1 only if address sent out by PC

* If it is MR in an IN or OUT instruction, the information is received in both W and Z registers.

Differences between MR and MW

	MR	*MW*
■ RD*, WR* Signals	RD* = 0, WR* = 1	RD* = 1, WR* = 0
■ Status of S1 and S0	S1 = 1, S0 = 0	S1 = 0, S0 = 1
■ AD_{7-0}	Floats during T2	Used for output of data during T2
■ Address sent out from	PC, SP, BC, DE, HL, or WZ	SP, BC, DE, HL, or WZ

Differences between MW and IOW

	MW	*IOW*
■ IO/M* signal	IO/M* = 0	IO/M* = 1
■ Data sent out from	any byte register, except IR	Accumulator
■ Address duplication on A_{15-8} and AD_{7-0}?	No	Yes
■ Address sent out from	SP, BC, DE, HL, or WZ	WZ only

Differences between MR and IOR

	MR	*IOR*
■ IO/M* signal	IO/M* = 0	IO/M* = 1
■ Data sent out from	any byte register, except IR	Accumulator only
■ Address duplication on A_{15-8} and AD_{7-0}?	No	Yes
■ Address sent out from	PC, SP, BC, DE, HL, or WZ	WZ only

Differences between IOR and IOW

	IOR	*IOW*
■ Status of S1 and S0	S1 =1, S0 = 0	S1 = 0, S0 = 1
■ RD*, WR* signals	RD* = 0, WR* =1	RD* = 1, WR* = 0
■ AD_{7-0}	floats during T2	used for output of data during T2

■ 13.4 MEMORY SPEED REQUIREMENT

At the end of the T2 state in a machine cycle, 8085 senses the Ready input pin. If it is logic 0, 8085 enters T_{wait} state, else it enters the T3 state. The Ready input is permanently tied to logic 1, if the memory chips and the I/O ports in the system are having compatible speed with 8085. Otherwise, appropriate number of wait states has to be generated by external circuitry. In fact, in the ALS kit the Ready pin is tied to logic 1. But then, how to conclude whether the memory chips are having compatible speeds or not? That is the topic of this section.

As an example, let us check up if the 27128A-20 16K × 8 EPROM chip can be used without any wait states with a 8085 operating at 320 nS clock period. For interfacing a memory chip, the most important timing parameters to be considered are t_{Acc}, t_{CE}, and t_{OE}. The memory read timing for a typical EPROM chip is shown in Fig. 13.17.

For 27128A-20 chip, the characteristics are as follows.

$$t_{Acc} = 200 \text{ nS max}$$
$$t_{CE} = 200 \text{ nS max}$$
$$t_{OE} = 75 \text{ nS max}$$

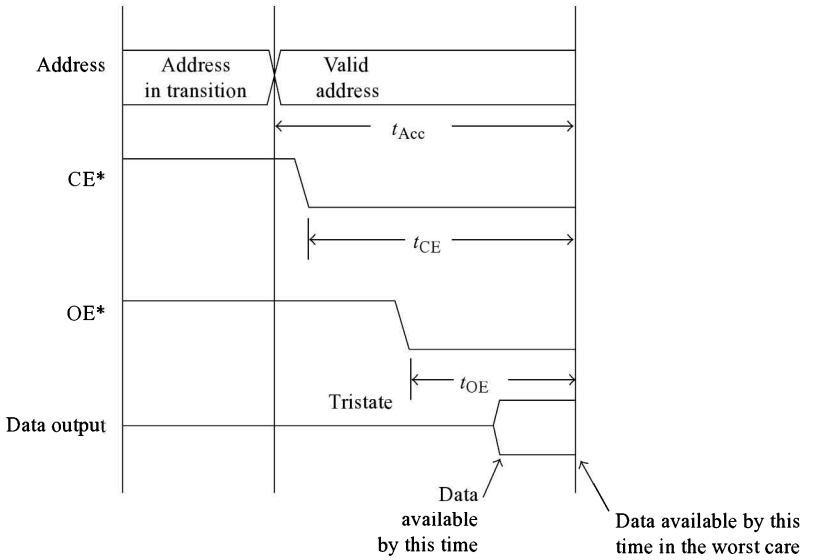


Fig. 13.17
Memory read timing for 27128A-20 16K × 8 EPROM

t_{Acc} = 200 nS means that, it takes a maximum of 200 nS to output the data after stable address is received by the 27128, under the following assumptions.

1. The memory chip would have been selected atleast 200 nS (t_{CE}) before the data is output by it.
2. The output enable line of the memory chip would have been activated atleast 75 nS (t_{OE}) before the data is output by it.

t_{CE} = 200 nS means that, it takes a maximum of 200 nS to output the data after the 27128 has been selected, under the following assumptions.

1. The memory chip would have received stable address atleast 200 nS (t_{Acc}) before the data is output by it.
2. The output enable line of the memory chip would have been activated atleast 75 nS (t_{OE}) before the data is output by it.

Generally address and chip select signals are received by a memory chip at about the same time. Of course, there will be some time difference depending on the propagation delays of the chips between the microprocessor and the memory chip.

t_{OE} = 75 nS means that, it takes a maximum of 75 nS to output the data after the 27128 output has been enabled, under the following assumptions.

1. The memory chip would have received stable address atleast 200 nS (t_{Acc}) before the data is output by it.
2. The memory chip would have been selected atleast 200 nS (t_{CE}) before the data is output by it.

Generally, it is the read signal that enables the output buffers of a memory chip to output the data. This signal is activated a little while after the address and chip select signals are received by the chip. The 27128 chip needs a maximum of 75 nS to output the data after the output of the chip is enabled.

The bus timing characteristics of MR machine cycle is shown in Fig. 13.18. The important parameters to be considered are t_{AD}, t_{LDR}, and t_{RD}. t_{AD} is the time interval between valid address on $A_{15\text{-}0}$ and valid data on $AD_{7\text{-}0}$. t_{LDR} is the time interval between trailing edge of ALE and valid data on $AD_{7\text{-}0}$ during a read operation. t_{RD} is the time interval between leading edge of RD* and valid data on $AD_{7\text{-}0}$.

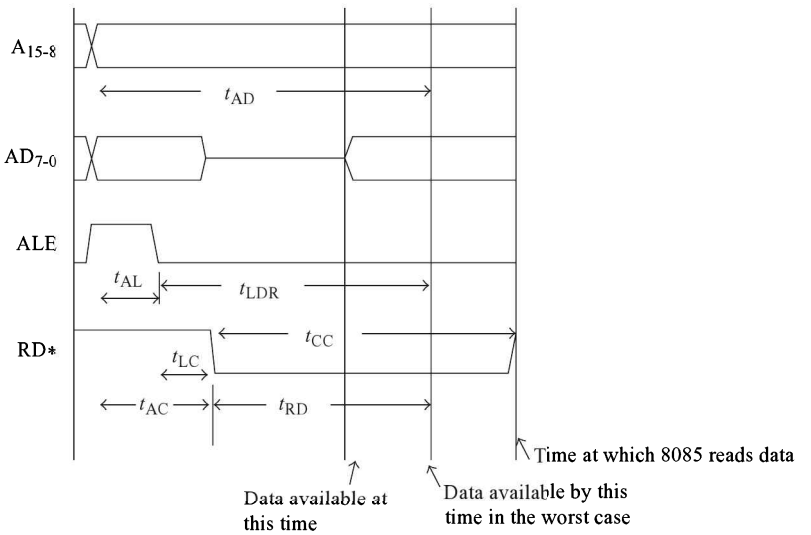


Fig. 13.18
Bus timing characteristics of MR machine cycle

The maximum values of these parameters in nanoseconds, as provided by Intel manuals, for 8085AH and 8085AH-2 are as shown in the following. In this table, T stands for cycle time and N stands for number of wait states.

	8085AH	***8085AH-2***
t_{AD}	$(2.5+N)T-225$	$(2.5+N)T-150$
t_{LDR}	$2T-180$	$2T-130$
t_{RD}	$(1.5+N)T-180$	$(1.5+N)T-150$

If we assume that there are no wait states, and T value as 320 nS (3.125 MHz) for 8085AH, and 200 nS (5 MHz) for 8085AH-2, we get maximum values of these parameters as:

	8085AH	***8085AH-2***
t_{AD}	575 nS	350 nS
t_{LDR}	460 nS	270 nS
t_{RD}	300 nS	150 nS

The other important parameters to be considered are t_{AL}, t_{AC}, and t_{LC}. t_{AL} is the time interval between valid address on $A_{15\text{-}0}$ and trailing edge of ALE. t_{AC} is the time interval between valid address on $AD_{15\text{-}0}$ and leading edge (1 to 0 transition) of RD* control signal during a read operation. t_{LC} is the time interval between trailing edge (1 to 0 transition) of ALE and leading edge of RD* control during a read operation.

The minimum values of these parameters in nanoseconds, as provided by Intel manuals, for 8085AH and 8085AH-2 are indicated as follows.

	8085AH	*8085AH-2*
t_{AL}	$0.5T-45$	$0.5T-50$
t_{AC}	$T-50$	$T-85$
t_{LC}	$0.5T-30$	$0.5T-40$

If we assume T value as 320 nS (3.125 MHz) for 8085AH, and 200 nS (5 MHz) for 8085AH-2, we get minimum values of these parameters as:

	8085AH	*8085AH-2*
t_{AL}	115 nS	50 nS
t_{AC}	270 nS	115 nS
t_{LC}	130 nS	60 nS

The other useful parameter to be considered is t_{CC}. It is the width of the RD* control signal in the logic 0 state during a read operation. The minimum value of this parameter in nanoseconds, as provided by Intel manuals, for 8085AH and 8085AH-2 are as shown.

	8085AH	*8085AH-2*
t_{CC}	$(1.5+N)T-80$	$(1.5+N)T-70$

If we assume that there are no wait states, and T value as 320 nS (3.125 MHz) for 8085AH, and 200 nS (5 MHz) for 8085AH-2, we get minimum value of this parameter as:

	8085AH	*8085AH-2*
t_{CC}	400 nS	230 nS

The 8085 reads the data on the AD_{7-0} lines when RD* makes 0 to 1 transition. Thus maximum value of t_{AD} should be ideally $t_{AC} + t_{CC}$. So for 8085AH working at 320 nS, maximum value of t_{AD} should be 270 nS + 400 nS = 670 nS. But practically, the data should be present on AD_{7-0} some time before the 0 to 1 transition of RD* signal. Thus Intel manuals specify a maximum value of 575 nS for t_{AD} when 8085AH is working at 320 nS.

Similarly, maximum value of t_{LDR} should be ideally $t_{LC} + t_{CC}$. So for 8085AH working at 320 nS, maximum value of t_{LDR} should be 130 nS + 400 nS = 530 nS. But practically, the data should be present on AD_{7-0} some time before the 0 to 1 transition of RD* signal. Thus Intel manuals specify a maximum value of 460 nS for t_{LDR} when 8085AH is working at 320 nS.

Similarly, maximum value of t_{RD} should be ideally t_{CC}. So for 8085AH working at 320 nS, t_{RD} should be 400 nS. But practically, Intel manuals specify a maximum value of 300 nS for t_{RD} when 8085AH is working at 320 nS.

Fig. 13.19 shows the actual interfacing details of 27128A-2 chip in the ALS kit. Using this figure and figures 13.17 and 13.18, we can assess the compatibility of this chip with 8085AH working with 320-nS clock period.

We start the calculations assuming that stable address and IO/M* signals are sent out by the 8085AH at time 0 nS. Then, ALE goes to 0 state at 115 nS (t_{AL}), and RD* is activated at 270 nS (t_{AC}).

13.4.1 Earliest Data Output Time Considering t_{ACC}

Address A_{13-8} is received by the 27128 from 8085AH via 74LS244 octal line driver, which has a propagation delay of 12 nS. Address A_{7-0} is received by the 27128 from 8085AH via 74LS373 octal latch,

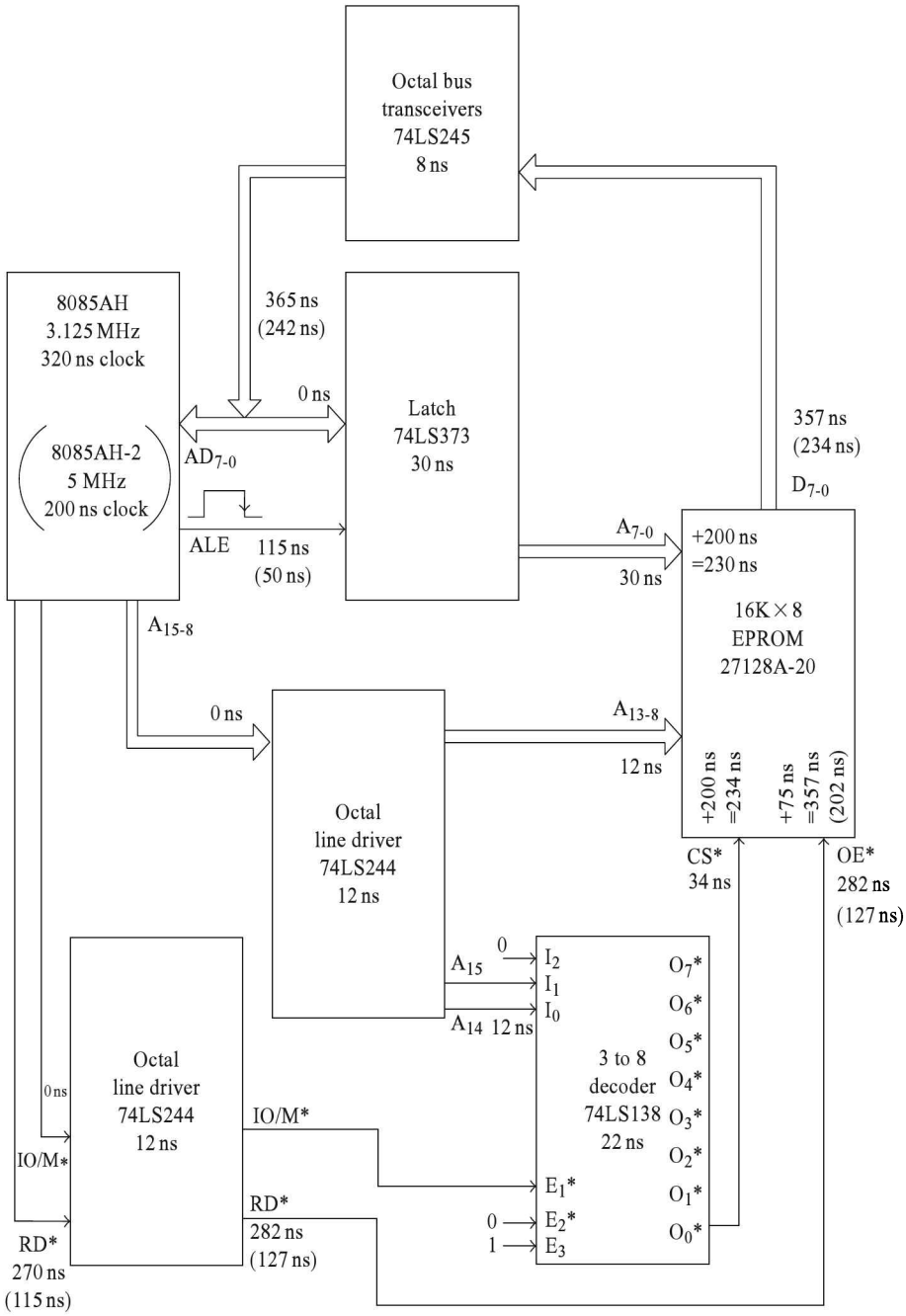


Fig. 13.19 Delays involved in accessing 27128 EPROM in ALS kit

which has a propagation delay of 30 nS. Thus, address A_{13-0} is received by 27128 at the end of 30 nS. Thus, the data can only come out on D_{7-0} pins of 27128 by 30 nS + t_{Acc} = 30 nS + 200 nS = 230 nS.

13.4.2 Earliest Data Output Time Considering t_{CE}

The 74138 receives A_{15-14} from 8085 via 74LS244 (octal line driver, with 12 nS delay). It simultaneously receives IO/M* signal from 8085 via 74LS244 (octal line driver, with delay of 12 nS). Then CS* signal is received by the 27128 from 74LS138 (3 to 8 decoder, with delay of 22 nS). Thus, CS* signal is received by 27128 at the end of 34 nS. Thus, the data can only come out on D_{7-0} pins of 27128 by 34 nS + t_{CE} = 34 nS + 200 nS = 234 nS.

13.4.3 Earliest Data Output Time Considering t_{OE}

The 8085AH activates RD* signal at 270 nS. This signal goes to OE* pin of 27128 via 74LS241 (octal line driver, with 12nS delay). Thus, OE* signal is received by 27128 at the end of 282 nS. Thus, the data can only come out on D_{7-0} pins of 27128 by 282 nS + t_{OE} = 282 nS + 75 nS = 357 nS.

From the discussion in the previous three paragraphs, it should be clear that the earliest data output time will be 357 nS, considering all the three parameters t_{ACC}, t_{CE}, and t_{OE}. In fact, t_{Acc} can be as large as 200 nS + (357 nS − 230 nS) = 327 nS, without affecting the time at which data comes out of memory chip. Similarly t_{CE} can be as large as 200 nS + (357 nS − 234 nS) = 323 nS, without affecting the time at which data comes out of the memory chip.

The 8085AH receives this data from 27128 via 74LS245 (octal bus transceiver, with delay of 8 nS). Thus, valid data is received by the 8085AH at 357 nS + 8 nS = 365 nS.

13.4.4 27128-20 Compatibility Check with 8085AH

Now let us perform memory compatibility check with respect to the parameters t_{AD}, t_{LDR}, and t_{RD}.

Compatibility with respect to t_{AD}: t_{AD} is the time interval between valid address on A_{15-0} and valid data on AD_{7-0}. For 8085AH working with a T state of 320 nS, it is a maximum of 575 nS. But in this case, the valid data is available in 365 nS itself. Thus the memory speed is compatible, with actually an excess time margin of 575 nS − 365 nS = 210 nS.

Compatibility with respect to t_{LDR}: t_{LDR} is the time interval between trailing edge of ALE and valid data on AD_{7-0} during a read operation. For 8085AH working with a T state of 320 nS, it is a maximum of 460 nS. The trailing edge of ALE occurs at 115 nS. Thus t_{LDR} in this case is only 365 nS − 115 nS = 250 nS. Thus, the memory speed is compatible, with actually an excess time margin of 460 nS − 250 nS = 210 nS.

Compatibility with respect to t_{RD}: t_{RD} is the time interval between leading edge of RD* and valid data on AD_{7-0}. For 8085AH working with a T state of 320 nS, it is a maximum of 300 nS. The 8085AH activates RD* signal at 270 nS and the valid data is available by 365 nS itself. Thus t_{RD} in this case is only 365 nS − 270 nS = 95 nS. Thus, the memory speed is compatible with actually an excess time margin of 300 nS − 95 nS = 205 nS.

Thus, 27128-20 EPROM chip is speed compatible with 8085AH. In fact, as per the discussion in the previous three paragraphs, there is an excess time margin of atleast 205 nS. Thus, the 8085AH can be interfaced with memory chips, which have the worst case specifications as shown in the following.

$$t_{Acc} = 327 \text{ nS} + 205 \text{ nS} = 532 \text{ nS}$$
$$t_{CE} = 323 \text{ nS} + 205 \text{ nS} = 528 \text{ nS}$$
$$t_{OE} = 75 \text{ nS} + 205 \text{ nS} = 280 \text{ nS}$$

In this case, as discussed in section (13.4.3), 327 nS is the largest t_{Acc} possible without affecting the time (357 nS) at which the data comes out of memory chip. Corresponding values for t_{CE} and t_{OE} are 323 nS and 75 nS respectively. 205 ns in the margin still available when 8085AH is interfaced with memory chips.

13.4.5 Assessing Compatibility of 27128-20 with 8085AH-2

The 8085AH-2 works with 200nS clock period. We start the calculations assuming that valid address, and IO/M* signals are sent out by the 8085AH-2 at time 0 nS. Then, ALE goes to 0 state at 50 nS (t_{AL}), and RD* is activated at 115 nS (t_{AC}).

Earliest data output time considering t_{Acc}: Address $A_{13\text{-}18}$ is received by the 27128 from 8085 via 74LS244 (octal line driver, with delay of 12 nS). Address $A_{7\text{-}0}$ is received by the 27128 from 8085 via 74LS373 (octal latch, with delay of 30 nS). Thus, address $A_{13\text{-}0}$ is received by 27128 at the end of 30 nS. Thus, the data can only come out on $D_{7\text{-}0}$ pins of 27128 by 30 nS + t_{Acc} = 30 nS + 200 nS = 230 nS.

Earliest data output time considering t_{CE}: The 74138 receives $A_{15\text{-}14}$ from 8085 via 74LS244 (octal line driver, with 12 nS delay). It simultaneously receives IO/M* signal from 8085 via 74LS241 (octal line driver, with delay of 12 nS). Then CS* signal is received by the 27128 from 74LS138 (3 to 8 decoder, with delay of 22 nS). Thus, CS* signal is received by 27128 at the end of 34 nS. Thus, the data can only come out on $D_{7\text{-}0}$ pins of 27128 by 30 nS + t_{CE} = 34 nS + 200 nS = 234 nS.

Earliest data output time considering t_{OE}: The 8085AH-2 activates RD* signal at 115 nS. This signal goes to OE* pin of 27128 via 74LS241 (octal line driver, with 12 nS delay). Thus, OE* signal is received by 27128 at the end of 127 nS. Thus, the data can only come out on $D_{7\text{-}0}$ pins of 27128 by 127 nS + t_{OE} = 127 nS + 75 nS = 202 nS.

From the discussion in the previous three paragraphs, it should be clear that the earliest data output time will be 234 nS, considering all the three parameters t_{ACC}, t_{CE}, and t_{OE}. In fact, t_{Acc} can be as large as 200 nS + (234 nS − 230 nS) = 204 nS, without affecting the time at which data comes out of memory chip. Similarly t_{OE} can be as large as 75 nS + (234 nS − 202 nS) = 107 nS, without affecting the time at which data comes out of memory chip.

The 8085AH-2 receives this data from 27128 via 74LS245 (octal bus transceiver with delay of 8 nS). Thus, valid data is received by the 8085AH-2 at 234 nS + 8nS = 242 nS.

Now let us perform memory compatibility check with respect to the parameters t_{AD}, t_{LDR}, and t_{RD}.

Compatibility with respect to t_{AD}: For 8085AH-2 working with a T state of 200 nS, it is a maximum of 350 nS. But in this case, the valid data is available in 242 nS itself. Thus the memory speed is compatible with actually an excess time margin of 350 nS − 242 nS = 108 nS.

Compatibility with respect to t_{LDR}: For 8085AH-2 working with a T state of 200 nS, it is a maximum of 270 nS. The trailing edge of ALE occurs at 50 nS. Thus t_{LDR} in this case is only 242 nS − 50 nS = 192 nS. Thus, the memory speed is compatible, with actually an excess time margin of 270 nS − 192 nS = 78 nS.

Compatibility with respect to t_{RD}: For 8085AH-2 working with a T state of 200 nS, it is a maximum of 150 nS. The 8085AH-2 activates RD* signal at 115 nS and the valid data is available by 242 nS itself. Thus, t_{RD} in this case is only 242 nS − 115 nS = 127 nS. Thus, the memory speed is compatible, with actually an excess time margin of 150 nS − 127 nS = 23 nS.

Thus, 27128-20 EPROM chip is speed compatible with 8085AH-2 also. In fact, as per the discussion in the previous three paragraphs, there is an excess time margin of atleast 23 nS. Thus, the 8085AH-2 can be interfaced with memory chips, which have the worst case specifications indicated as follows:

$$t_{ACC} = 204 \text{ nS} + 23 \text{ nS} = 227 \text{ nS}$$
$$t_{CE} = 200 \text{ nS} + 23 \text{ nS} = 223 \text{ nS}$$
$$t_{OE} = 107 \text{ nS} + 23 \text{ nS} = 130 \text{ nS}$$

In this case, 204 nS is the largest t_{Acc} possible without affecting the time (234 nS) at which the data comes out of memory chip. Corresponding values for t_{CE} and t_{OE} are 200 nS and 107 nS respectively. 23 ns in the margin still available when 8085AH-2 is interfaced with memory chips.

■ 13.5 WAIT STATE GENERATION

The present day memory and peripheral chips are fast enough for a 8085 working at 3 MHz. So wait states are generally not needed at all. If 8085AH-2 is used, which can work at 5 MHz, there may be a need to insert one wait state, between T2 and T3. The circuit shown in Fig. 13.20 easily achieves this.

The circuit uses two D-type positive edge-triggered flip flops, with active low Reset inputs. At the beginning of T1, ALE goes high and causes Q1 to go high. As Q1 and D2 are connected, D2 remains high throughout T1. The positive edge of T2 clock causes Q2* to become 0. This is connected to ready

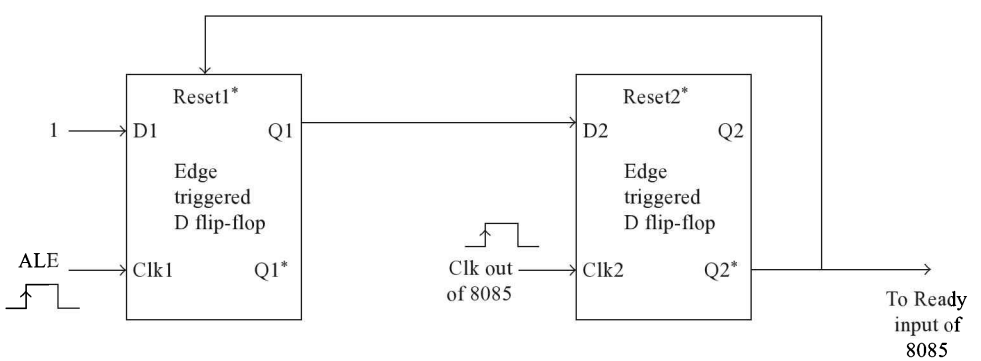


Fig. 13.20 Circuit to insert a wait state between T2 and T3

input of 8085. Thus, ready input remains 0 throughout the T2 state. So 8085 enters T_{wait}, instead of T3, after T2. Also, when Q2* becomes 0, Reset1* becomes 0, thus making Q1 output as 0 throughout T2 state. The positive edge of T_{wait} causes Q2* to become 1. This makes ready input as 1, and so after T_{wait} state, 8085 enters the T3 state. Thus, one wait state is inserted by the circuit in Fig. 13.20 between T2 and T3. This can be visualized from the waveforms indicated in Fig. 13.21.

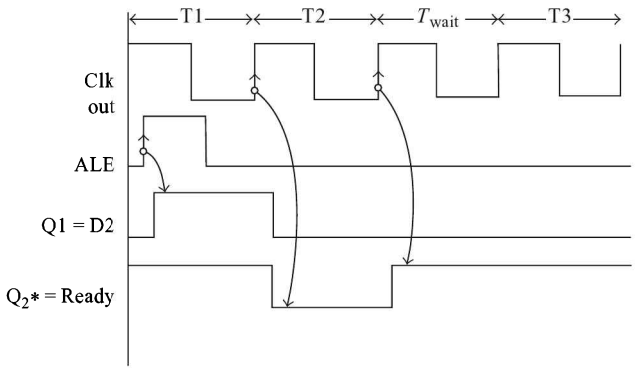


Fig. 13.21
Waveforms indicating generation of wait state

1. What is the difference between the programmer's view of 8085 and architecture of 8085?
2. With a neat diagram explain the architecture of 8085 needed for instruction execution.
3. Explain the role of IR, Temp, W and Z registers in the architecture of 8085.
4. Indicate with reason, whether the following registers are connected to the internal bus in a unidirectional or bi-directional way.

   a. Accumulator
   b. Temp
   c. IR
   d. Flags
   e. W
   f. D

5. Mention the steps involved in an instruction cycle and explain the steps with an example.
6. Explain the terms clock cycle (T state), machine cycle, and instruction cycle.
7. List the machine cycles, the registers involved, and the total number of clock cycles needed in the fetching and execution of the following instructions (check your answer with the information provided in the appendix).

a. MVI B, 25H;	b. STAX D;	c. LXI B, 1234H;	d. XRA B;
e. JNC 1234H;	f. CALL 1234H;	g. DAD D;	h. POP B;
i. RST 5;	j. PCHL;	k. XTHL;	l. SPHL;
m. XCHG;	n. RNC;	o. SHLD 1234H;	p. OUT 25H.

8. Give two examples for each of the following.

   a. Instructions that need only OF with 4 *T* states;
   b. Instructions that need only OF with 6 *T* states;
   c. Instructions that need OF and MR;
   d. Instructions that need OF, MR, and MR;
   e. Instructions that need OF, MR, and MW.

9. Give atleast one example for each of the following.

   a. Instructions that need OF, MR, MR, MW, and MW;
   b. Instructions that need OF, MR, and IOR;
   c. Instructions that need OF, BI, and BI;
   d. Instructions that need OF, MW, and MW.

10. Compare the following machine cycles.

    a. OF with MR;
    b. MR with IOR;
    c. MW with IOW.

11. Explain the terms t_{Acc}, t_{CE}, and t_{OE} for a memory chip.
12. Explain the following terms with reference to a MR cycle.

    a. t_{AD};
    b. t_{LDR};
    c. t_{RD};
    d. t_{AL};
    e. t_{AC};
    f. t_{LC}.

13. Suppose a memory chip has the following specifications.

    t_{Acc} = 350 nS
    t_{CE} = 360 nS
    t_{OE} = 180 nS

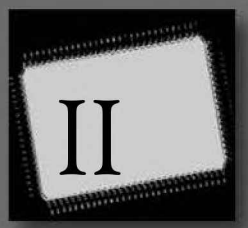
    Can this chip be interfaced with (assume interface circuit of Fig. 13.19):

    a. 8085AH with a *T* state of 350 nS?
    b. 8085AH-2 with a *T* state of 220 nS?

14. Calculate the number of wait states needed if 8085AH-2 working with a *T* state of 200 nS is required to be interfaced with a memory chip having the following specifications. Assume interface circuit of Fig. 13.19.

    t_{Acc} = 550 nS
    t_{CE} = 500 nS
    t_{OE} = 220 nS

15. Explain with a neat diagram a circuit to insert one wait state between T2 and T3 states in a machine cycle.



Assembly Language Programs

Chapter Heads

14 Simple Assembly Language Programs
15 Use of PC in Writing and Executing 8085 Programs
16 Additional Assembly Language Programs
17 More Complex Assembly Language Programs

INTRODUCTION

In this part, we will mainly discuss the programs for a number of interesting problems. This part comprises of Chapters 14 to 17. In Chapter 14, we deal with simple 8085 assembly language programs. In Chapter 15, we deal with the use of PC in writing and executing 8085 programs. In Chapter 16, we discuss additional assembly language programs, while Chapter 17 discusses comparatively more complex assembly language programs.

14 Simple Assembly Language Programs

■ Exchange 10 bytes
• *Entry of data and code*
• *Executing the program and checking result*
■ Add two multi-byte numbers
■ Add two multi-byte BCD numbers
■ Block movement without overlap
■ Block movement with overlap
• *Approach methodology*
■ Add *N* numbers, of size 8 bits
• *Monitor routines*
■ Check the fourth bit of a byte
■ Subtract two multi-byte numbers
■ Multiply two numbers of size 8 bits
• *Trace of the program*
■ Divide a 16-bit number by an 8-bit number
■ Questions

This chapter deals with various assembly language programs, such as programs to exchange 10 bytes, to add/subtract two multi-byte numbers, to add two multi-byte BCD numbers, to multiply two 8-bit numbers, and to divide a 16-bit number by an 8-bit number. These assembly language programs will be manually translated using the opcode information provided in the appendix.

■ 14.1 EXCHANGE 10 BYTES

Write an 8085 assembly language program to exchange 10 bytes of data stored from location X with 10 bytes of data stored from location Y.

Flowchart for the program
Flowchart for solving the problem is shown in Fig. 14.1.

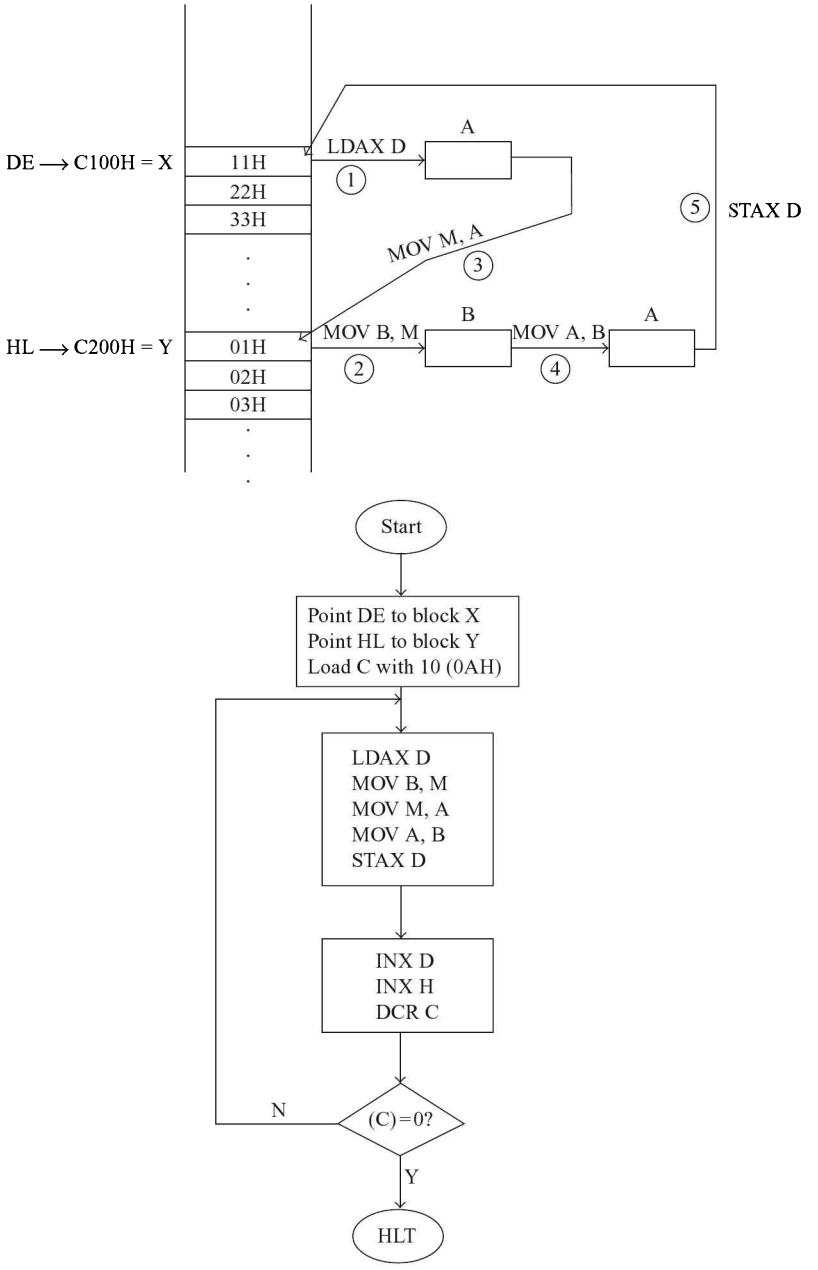


Fig. 14.1 Flowchart for exchange of 10 bytes

Program to exchange 10 bytes

```
; FILE NAME C:\ALS\XCHG.ASM
  ; Suppose we want to make some changes to the source file. If a print out
;of the program informs us about the file name for the program on the disk,
;it becomes easy to search the file and make modifications to the program.

   ;8085 ALP TO INTERCHANGE 10 BYTES OF DATA STORED FROM
   ;LOCATION X WITH 10 BYTES OF DATA STORED FROM LOCATION Y

   ORG C100H
  X: DB 11H, 22H, 33H, 44H, 55H, 66H, 77H, 88H, 99H, AAH

   ORG C200H
  Y: DB 01H, 02H, 03H, 04H, 05H, 06H, 07H, 08H, 09H, 0AH

;For manual translation store 11H,22H etc starting at location C100H.
;Similarly, the following program after manual translation should be stored
;from location C000H. ORG, DB are Assembler directives and are explained in
;detail in the next chapter.

   ORG C000H

   LXI D, X ; Load DE with C100H (address X)
   LXI H, Y ; Load HL with C200H (address Y)
   MVI C, 0AH; Load C with 0AH (C used as down counter)

   ; The instructions from here to JNZ LOOP perform exchange
   ; of bytes at memory locations pointed by DE and HL

LOOP:    LDAX D     ; Load A from memory pointed by DE
         MOV B, M   ; Load B from memory pointed by HL

         MOV M, A   ; Store A in memory pointed by HL
         MOV A, B
         STAX D     ;;Store B in memory pointed by DE

;In the convention followed in this book, a single ';' after an instruction
;provides comments for the instruction. If there are N number of ';' after an
;instruction, it provides comments for N instructions preceding the N number
;of semicolons.

      INX D
      INX H      ;;Increment address pointers DE and HL
      DCR C      ; Decrement C
      JNZ LOOP   ; If not zero jump to LOOP

      HLT        ; Stop when all the bytes are exchanged
```

14.1.1 Entry of Data and Code

The data and code to be entered on the kit for the previous program is shown in the following.

Entry of data: In the previous program, X is symbolic memory location C100H, and data is stored from C100H as follows.

Address	*Data*
C100	11H
C101	22H
C102	33H
C103	44H
C104	55H
C105	66H
C106	77H
C107	88H
C108	99H
C109	AAH

Similarly, Y is symbolic memory location C200H, and data is stored from C200H as follows.

Address	*Data*
C200	01H
C201	02H
C202	03H
C203	04H
C204	05H
C205	06H
C206	07H
C207	08H
C208	09H
C209	0AH

Entry of code: In the previous program, code is stored from C000H as follows with addresses and code in hexadecimal.

Label	*Mnemonic*	*Operand/s*	*Address*	*Hexcode*
	LXI D,	X	C000	11 00 C1
	LXI H,	Y	C003	21 00 C2
	MVI C,	0AH	C006	0E 0A
LOOP:	LDAX	D	C008	1A
	MOV	B, M	C009	46
	MOV	M, A	C00A	77
	MOV	A, B	C00B	78
	STAX	D	C00C	12
	INX	D	C00D	13
	INX	H	C00E	23
	DCR	C	C00F	0D
	JNZ	LOOP	C010	C2 08 C0
	HLT		C013	76

It can be noticed from this that LOOP is a symbolic memory location, which is actually memory location with address C008H. Thus, JNZ LOOP is manually translated as C2 08 C0, where C2 is code for JNZ and 08 C0 is C008H stored in memory in byte reversal form.

14.1.2 Executing the Program and Checking Result

When the program is run by typing on the kit 'Go C 0 0 0 Exec', we see the display 'E' in the address field of the display. At this point the 8085 has halted processing because of the HLT instruction execution. To check the results after program execution, we have to first of all reset the 8085 by typing 'Reset' key. Then the monitor program gains control over the kit, and we can see the contents of C100 to C109, and C200 to C209.

Alternatively, we can end the program with 'RST 1' (code = CF) instruction instead of HLT instruction, when we are using ALS kit. Execution of RST 1 instruction on ALS kit results in the monitor program gaining control over the kit. So we are now in a position to see the results in memory. Thus we can avoid the step of resetting the 8085. The reader must note that with other kits he may have to terminate the program with say, RST 5, to transfer control to the monitor program. Refer to kit manual to obtain correct information.

■ 14.2 ADD TWO MULTI-BYTE NUMBERS

Write a 8085 assembly language program to add two multi-byte numbers. The numbers are stored from locations X and Y in byte reversal form. The size in bytes of the multi-byte numbers is given in the location, SIZE. The result is to be stored in location Z, in byte reversal form, using 1 byte more than the size of multi-byte numbers.

Flowchart for the program
Flowchart for solving the problem is shown in Fig. 14.2.

Program to add two multi-byte numbers

```
;FILE NAME C:\ALS\ADLRGBYT.ASM

;8085 ALP TO ADD 2 MULTI BYTE NUMBERS.
;THE NUMBERS ARE STORED FROM LOCATIONS X AND Y.
;AT X AND Y, THE LS BYTE OF MULTI BYTE NUMBER IS PRESENT.
;THE SIZE IN BYTES OF THE MULTI BYTE NUMBERS IS GIVEN IN LOCATION SIZE
;THE RESULT IS STORED FROM LOCATION Z, USING SIZE+1 BYTES.

            ORG C050H
SIZE        DB 03H

            ORG C100H
X:          DB 12H, 34H, 56H

            ORG C200H
Y:          DB 56H, 78H, ACH

            ORG C000H
Z:          EQU C300H; Z is equal to C300H

;Thus wherever we come across Z, we replace it with C300H

      LXI B, X ;Load BC with C100H (address X)
      LXI H, Y ;Load HL with C200H (address Y)
      LXI D, Z ;Load DE with C300H (address Z)
```

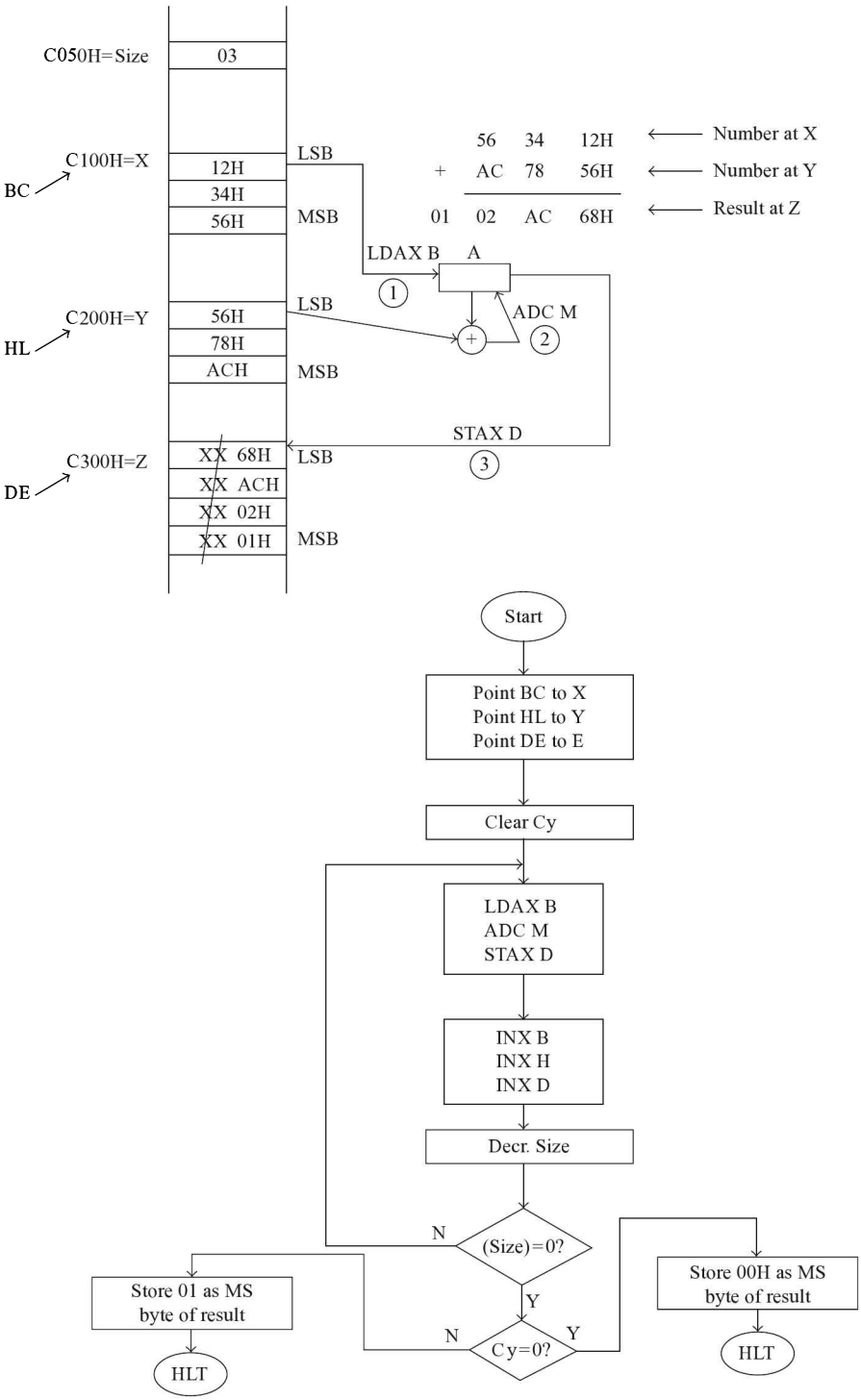


Fig. 14.2 Flowchart for adding two multi-byte numbers

```
        STC
        CMC     ;;Reset Cy flag to 0
;The instructions from here to JNZ LOOP perform addition with carry of the
;bytes pointed by BC and HL, and stores the result at memory pointed by
;DE. BC, HL, and DE pointers are incremented, and contents of location SIZE
;is decremented.
LOOP:   LDAX B   ;Load A from memory pointed by BC
        ADC M    ;Add to A with Cy memory contents pointed by HL
        STAX D   ;Store result in memory pointed by DE

        INX B
        INX H
        INX D    ;;;Increment the pointers BC, HL, and DE

        LDA SIZE
        DCR A
        STA SIZE ;;;Decrement contents of memory location SIZE
        JNZ LOOP ;If result of decrement is non zero jump to LOOP
;When we are out of this loop A contents is 00H.

        JNC SKIP ;Jump to SKIP if Cy = 0
        INR A

SKIP:   STAX D ;; Store 00 or 01 in memory pointed by DE, based on Cy

        HLT      ;Alternatively terminate the program with RST 1
```

Notice that INX B, INX H, and INX D instructions do not affect any flags. Also note that DCR A instruction affects all flags except Cy. Thus, when we execute JNC SKIP later on in the program, the jump takes place based only on the value of the Cy flag after execution of the ADC M instruction. In fact, in view of such requirements only, the designers of 8085 have implemented INX instructions such that flags are not affected, and DCR/INR instructions such that Cy flag is not affected.

In this program, we faced the problem of shortage of registers, although seven registers are provided in 8085. So we had to make use of memory location SIZE. In Motorola 6800, with only two registers inside, we do not face any problem! This is because of the powerful indexed addressing mode provided in 6800. See chapter on 6800 microprocessor for details.

■ 14.3 ADD TWO MULTI-BYTE BCD NUMBERS

Write an 8085 assembly language program to add two multi-byte BCD numbers. The numbers are stored in locations X and Y in byte reversal form. The size in bytes of the multi-byte numbers is given in the location, SIZE. The result is to be stored in location Z, in byte reversal form, using 1 byte more than the size of multi-byte numbers.

Flowchart for the program

Flowchart for solving the problem is shown in Fig. 14.3.

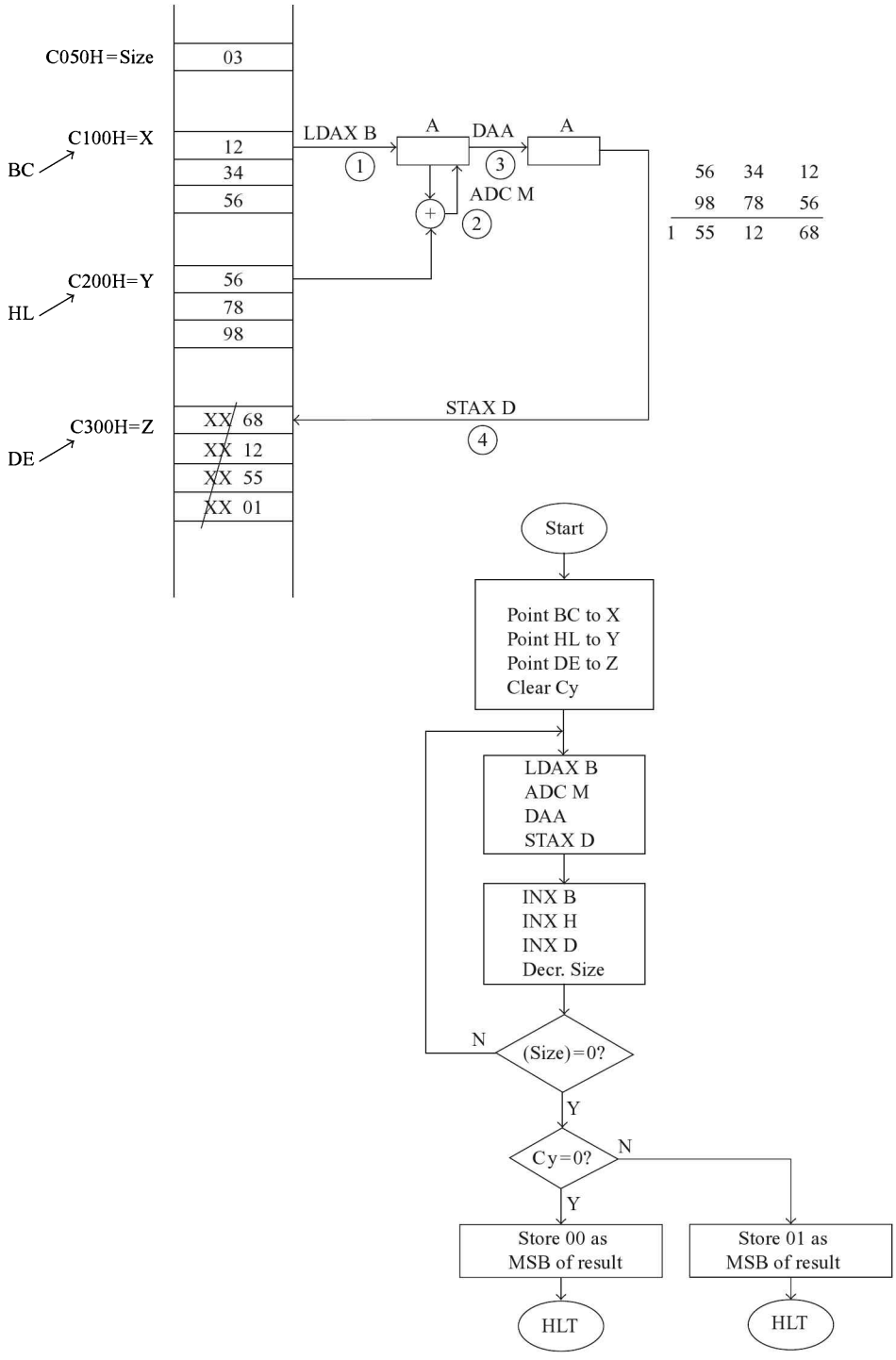


Fig. 14.3 Flowchart for adding two multi-byte BCD numbers

Program to add two multi-byte BCD numbers

```
;FILE NAME C:\ALS\ADLRGBCD.ASM
;8085 ALP TO ADD 2 MULTI BYTE BCD NUMBERS.
;THE NUMBERS ARE STORED FROM LOCATIONS X AND Y.
;AT X AND Y, THE LS BYTE OF MULTI BYTE NUMBER IS PRESENT.
;THE SIZE IN BYTES OF THE MULTI BYTE NUMBERS IS GIVEN IN LOCATION SIZE
;THE RESULT IS STORED FROM LOCATION Z, USING SIZE+1 BYTES.

            ORG C050H
SIZE   DB 03H

            ORG C100H
X:          DB 12H, 34H, 56H

            ORG C200H
Y:          DB 56H, 78H, 98H

            ORG C000H
Z:          EQU C300H

      LXI B, X  ;Load BC with C100H (address X)
      LXI H, Y  ;Load HL with C200H (address Y)
      LXI D, Z  ;Load DE with C300H (address Z)

      STC
      CMC       ;;Clear Carry flag

;The instructions from here to JNZ LOOP perform decimal addition with carry
;of a byte pointed by BC with a byte pointed by HL. The result is stored in
;memory pointed by DE. BC, HL, and DE pointers are incremented, and contents
;of location SIZE is decremented.
  LOOP:     LDAX B   ;Load A from memory pointed by BC
            ADC M    ;Add with Cy memory contents pointed by HL
            DAA      ;Perform decimal adjustment
            STAX D   ;Store result in memory pointed by DE

            INX B
            INX H
            INX D    ;;;Increment the pointers BC, HL, and DE by 1

            LDA SIZE
            DCR A
            STA SIZE ;;;Decrement contents of location SIZE
            JNZ LOOP  ;If result of decrement is nonzero jump to LOOP

  ;When we are out of this loop, contents of A will be 00H

            JNC SKIP
            INR A

  SKIP:     STAX D ;;;Store 00 or 01 in memory pointed by DE based on Cy

            HLT      ;Alternatively terminate with RST 1
```

■ 14.4 BLOCK MOVEMENT WITHOUT OVERLAP

Write an 8085 assembly language program to perform block movement. The blocks are assumed to be non-overlapping. The block starting at location X is to be moved to the block starting at Y. The block size is provided in the location, SIZE.

Flowchart for the program

Flowchart for solving the problem is shown in Fig. 14.4.

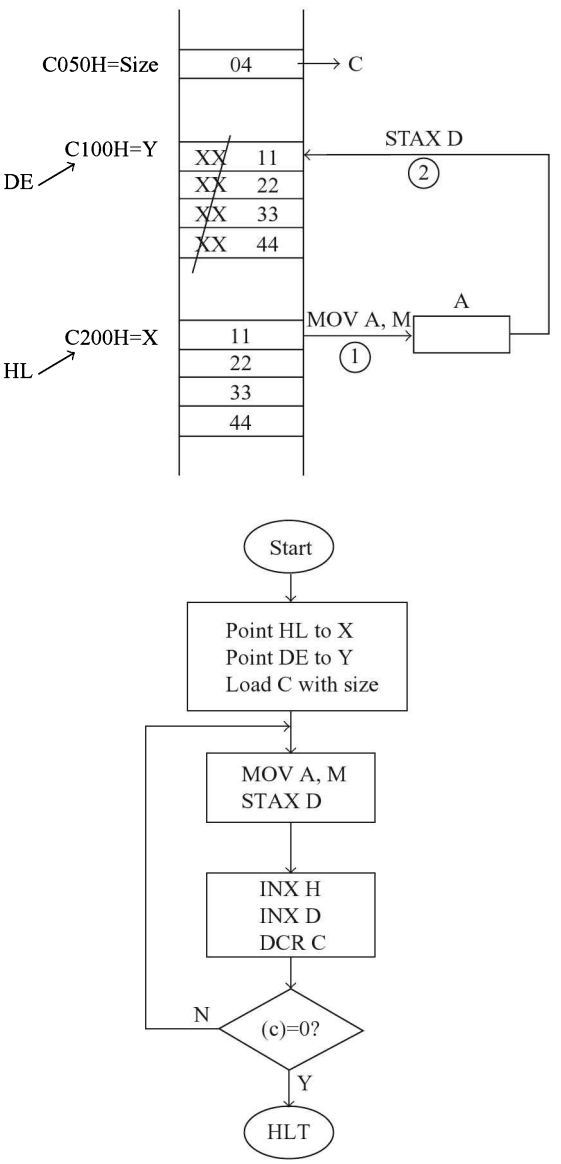


Fig. 14.4
Performing block movement without overlap

Program to perform block movement without overlap

```
;FILE NAME C:\ALS\BLKMOV.ASM

;8085 ALP TO PERFORM BLOCK MOVEMENT OF A BLOCK STARTING
;FROM LOCATION X, TO THE BLOCK STARTING FROM LOCATION Y.
;SIZE OF THE BLOCK PROVIDED IN LOCATION SIZE.

  ;IT WORKS ONLY WHEN
      ;a. THERE IS NO OVERLAP OF BLOCKS
      ;b. THERE IS OVERLAP WITH ADDRESS X LARGER THAN ADDRESS Y

            ORG C050H
  SIZE:     DB 04H

            ORG C200H
  X:        DB 11H, 22H, 33H, 44H

            ORG C000H
  Y:        EQU C100H

            LDA SIZE
            MOV C, A ;Load C with contents of location SIZE

            LXI H, X ;Load HL with C200H (address of X)
            LXI D, Y ;Load DE with C100H (address of Y)

;The instructions from here to JNZ LOOP perform movement of the byte pointed
;by HL to memory pointed by DE. HL and DE pointers are incremented. Counter C
;is decremented.

  LOOP:     MOV A, M
            STAX D   ;;Move the byte pointed by HL to memory pointed by DE

            INX H
            INX D
            DCR C    ;;;Increment the pointers HL and DE. Decrement C.
            JNZ LOOP   ;Jump to LOOP if result after decrement is nonzero

            HLT        ;Alternatively terminate with RST 1
```

■ 14.5 BLOCK MOVEMENT WITH OVERLAP

Write an 8085 assembly language program to perform block movement. The block starting at location X is to be moved to the block starting at Y. The block size is provided in the location, SIZE. The program should work irrespective of overlap of blocks or not.

14.5.1 Approach Methodology

In this program we have used C register as a down counter to store source block on the stack. We have used B register as a down counter to pop out the source block on the stack to the destination block. The program works even if there is overlap of blocks.

We load B and C registers with the size of the block. Then load HL and DE with the starting address of source block and destination block, respectively.

Then the byte pointed by HL is moved to A. It is then pushed above the stack top. Even flags register is pushed on the stack top because of the execution of PUSH PSW. But we do not care about it. Incrementing the pointers HL and DE, and decrementing the counter C is done. If C contents are not yet zero, we repeat the operations indicated in this paragraph.

Thus when we come out of the loop, the source block is stored on the stack. Of course, the stack size would have become twice the block size, because of pushing flags register also every time. The DE register will now be pointing to the next location after the last location in the destination block.

Now decrement pointer DE. Pop the stack top to A and flags. Store A value in memory pointed by DE. Just ignore the flags value popped out. Now decrement the counter B. If B contents are not yet zero, we repeat the operations indicated in this paragraph.

Halt when we come out of this loop, as the block movement is over.

Flowchart for the program

Flowchart and the method for solving the problem is shown in Fig. 14.5.

Program to perform block movement even with overlap

```
;FILE NAME C:\ALS\BLKMOV2.ASM

;8085 ALP TO PERFORM BLOCK MOVEMENT OF A BLOCK STARTING
;FROM LOCATION X, TO THE BLOCK STARTING FROM LOCATION Y.

;PROGRAM WORKS WHETHER THERE IS OVERLAP OR NOT OF THE BLOCKS.

            ORG C050H
  SIZE:     DB 04H

            ORG C100H
  X:        DB 11H, 22H, 33H, 44H

            ORG C102H
  Y:        DB 33H, 44H, 55H, 66H

            ORG C000H
            LDA SIZE
            MOV B, A
            MOV C, A ;;;Load B and C registers with block size

            LXI H, X   ;Load HL with C100H (address X)
            LXI D, Y   ;Load DE with C102H (address Y)

      ;For the block size of 4, there will be overlap of blocks

;The instructions from here to JNZ LOOP1 perform pushing the byte pointed by
;HL to the stack, and then incrementing the pointers HL and DE, and
;decrementing the counter C.

  LOOP1:    MOV A,M
            PUSH PSW ;;Push the byte pointed by HL on to the stack
                     ;Flags is also pushed on stack, but we don't care
            INX H
            INX D     ;;Increment the pointers HL and DE
```

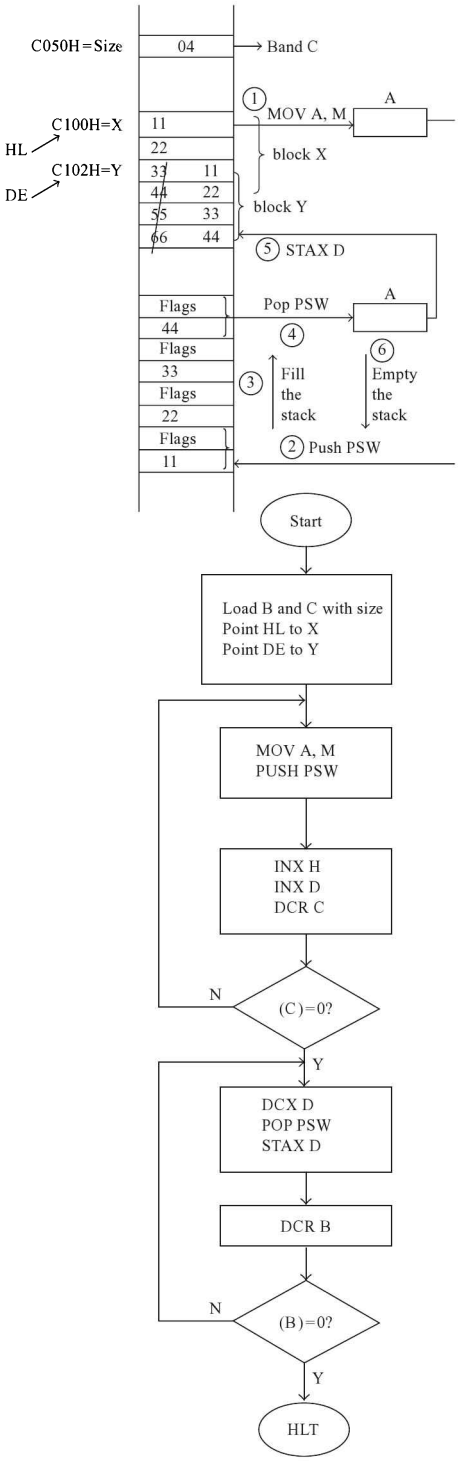


Fig. 14.5 Performing block movement even with overlap

```
            DCR C       ;Decrement C
            JNZ LOOP1   ;Jump to LOOP1 if C is not decremented to 00H

    :When are out of this loop, DE will be pointing to the next location
  ;after end of destination block

;The instructions from here to JNZ LOOP2 perform decrementing the pointer DE
;and then popping a byte from the stack and storing in memory pointed by DE.
;Finally decrements the counter B.

  LOOP2:  DCX D       ;Decrement the pointer DE
          POP PSW
          STAX D     ;;Pop the byte from stack top to memory pointed by DE
          DCR B       ;Decrement B
          JNZ LOOP2   ;Jump to LOOP2 if B is not decremented to 00H

          HLT
```

■ 14.6 ADD *N* NUMBERS OF SIZE 8 BITS

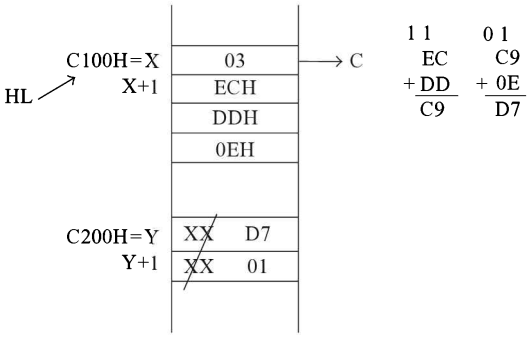
Write an 8085 assembly language program to perform addition of N 1-byte numbers. N value is provided in location X. From location X+1, the N numbers are stored. Store the result in location Y and Y+1. Also display the result in the address field.

Flowchart for the program

Flowchart for solving the problem is shown in Fig. 14.6.

Program to add N bytes

```
;FILE NAME C:\ALS\ADDNBYT.ASM

;8085 ALP TO ADD N ONE BYTE NUMBERS. N IS STORED AT LOCATION X.
;FROM X+1, THE NUMBERS ARE STORED. RESULT STORED IN LOCATIONS Y AND Y+1.
;ALSO DISPLAY THE RESULT IN THE ADDRESS FIELD.

            ORG C100H
  X:        DB 03H, ECH, DDH, 0EH

            ORG C000H

  Y:        EQU C200H
  CURAD:    EQU FFF7H ;Applicable only if ALS kit is being used
  UPDAD:    EQU 06BCH ;Applicable only if ALS kit is being used

            MVI B, 00H ;Initialise B with 00H.
            LXI H, X
            MOV C, M ;;Load C with number of bytes to be added
            DCR C    ;Decrement C. C now indicates the number of additions
                     ;to be performed
            INX H
            MOV A,M  ;;Load A with the first byte
```



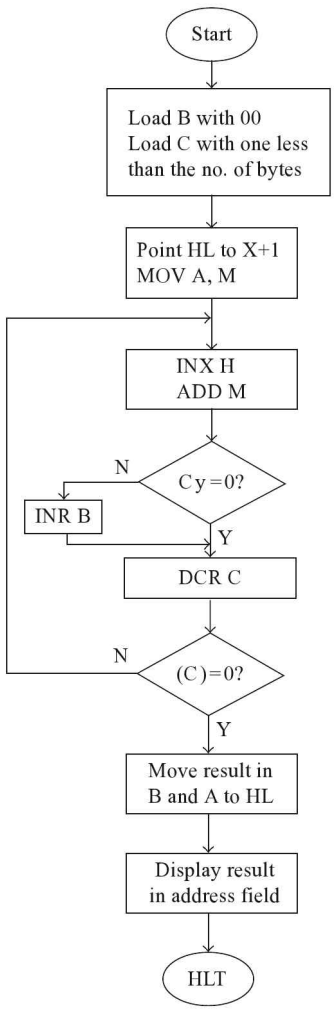


Fig. 14.6
Addition of *N* 1-byte numbers

```
;The instructions from here to JNZ AGAIN performs the following. It adds the
;next byte to A. B will be incremented by 1 if there is Carry. Decrements the
;counter C, and if nonzero repeats the operations.

  AGAIN:     INX H
             ADD M       ;;Add to A the next byte

             JNC NOINRB
             INR B       ;;B is incremented by 1 if there is Cy

  NOINRB:    DCR C        ;Decrement C
             JNZ AGAIN    ;If not zero jump to AGAIN

;When we come out of the loop, B will have MS byte of sum and A will have LS
;byte of sum

             MOV H, B
             MOV L, A   ;;Load HL with the sum in B and A registers
             SHLD Y      ;Store the sum in word location Y
             SHLD CURAD
             CALL UPDAD ;;Display the result in address field

             HLT         ;Make sure to terminate with HLT instead of RST 1
```

In the previous program, suppose we terminate the program with RST 1 instead of HLT. Then the program displays the result in the address field, and the next moment returns control to monitor program, which displays the sign on message. Thus, the user fails to see the result in the address field. If the program is terminated with HLT, the processor halts after the program displays the result in the address field. Thus, the result is displayed till the processor is reset.

14.6.1 Monitor Routines

Any 8085 microprocessor kit generally provides a number of utility routines, which can be called by the user. These routines form part of the monitor program in the EPROM/ROM of the kit. These routines simplify program development. As a general rule, the user should save all the registers of interest before calling a monitor routine. The registers should be restored with original values after returning from the monitor routine.

UPDAD routine: In this program, we have used the monitor routine UPDAD, which is used to update the address field on the kit. The routine displays the contents of the word at symbolic location CURAD.

On the ALS kit UPDAD routine is at address 06BCH, and CURAD is word location FFF7H. Thus, if we have 78H in location FFF7H and 56H at location FFF8H, execution of 'CALL UPDAD' instruction results in display of '5678' in the address field. CALL UPDAD is coded as CD BC 06.

In addition, if B register content is 00H, there will not be any dot at the end of the address field. If B contents are 01H, a dot will be displayed at the end of the address field.

The contents of the registers A, B, C, D, E, H, L, and flags are all altered on executing CALL UPDAD.

The other common routines like UPDDT, RDKBD are described later.

■ 14.7 CHECK THE FOURTH BIT OF A BYTE

Write an 8085 assembly language program to check whether the fourth bit of a byte at location X is a 0 or a 1. If 0, store 00 at location Y. Else, store FF at Y. Display the number and the result in the address field.

Flowchart for the program

Flowchart for solving the problem is shown in Fig. 14.7.

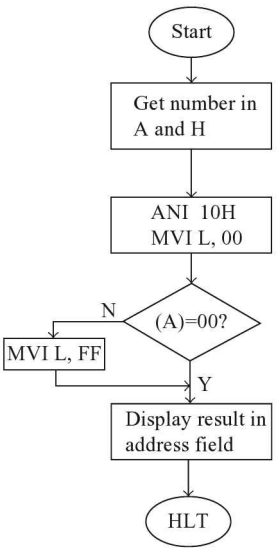


Fig. 14.7
To check the fourth bit of a given byte

Program to check the fourth bit of a given byte

```
;FILE NAME C:\ALS\CHEKBIT.ASM

  ;8085 ALP TO CHECK WHETHER THE 4TH BIT OF A BYTE STORED AT
  ;LOCATION X IS 0 OR 1. IF 0, STORE 00 AT LOCATION Y. ELSE, STORE
  ;FF AT Y. DISPLAY THE NUMBER AND THE RESULT IN THE ADDRESS FIELD.

            ORG C100H
  X:        DB 35H

            ORG C000H
  CURAD:    EQU FFF7H
  UPDAD:    EQU 06BCH

            LDA X
            MOV H, A   ;;Load A and H from contents of location X.
            MVI L, 00H ;Load L with 00.
            ANI 10H    ;Reset all bits to 0, except 4th bit.
            JZ OUTZERO ;If result is 00H, jump to OUTZERO.

            MVI L,FFH  ;If result is non zero, load L with FF.
```

```
  ;Thus at this point, L will have 00 or FF depending on bit 4.
OUTZERO:    SHLD CURAD
            CALL UPDAD ;;Display the byte, and 00 or FF, in address field.
            HLT
```

■ 14.8 SUBTRACT TWO MULTI-BYTE NUMBERS

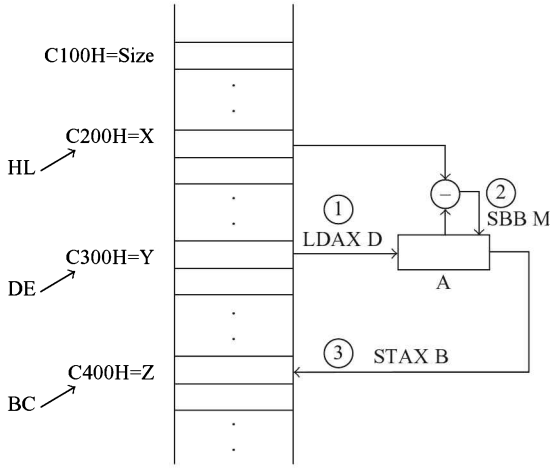
Write an 8085 assembly language program to perform subtraction of two multi-byte numbers. The number of bytes in these multi-byte numbers is stored in the location, SIZE. The numbers are stored in locations starting from X and Y. The number starting at X is subtracted from the number starting at Y. Result is stored in locations starting from Z.

Flowchart for the program

Flowchart for solving the problem is shown in Fig. 14.8.

Program to subtract two multi-byte numbers

```
;FILE NAME C:\ALS\SBLRGBIN.ASM

;8085 ALP TO SUBTRACT TWO MULTI BYTE BINARY NUMBERS. THE NUMBER OF BYTES IN
;THESE MULTI BYTE NUMBERS IS STORED AT LOCATION SIZE. THE NUMBERS ARE STORED
;IN LOCATIONS STARTING FROM X AND Y. THE NUMBER STARTING AT X IS SUBTRACTED
;FROM THE NUMBER STARTING AT Y. RESULT IS STORED IN LOCATIONS STARTING FROM
;Z.

;IF THE NUMBER STARTING AT LOCATION X IS THE LARGER ONE, THE RESULT WILL BE
;IN 2'S COMPLEMENT FORM.

            ORG C000H

  SIZE:     EQU C100H
  X:        EQU C200H
  Y:        EQU C300H
  Z:        EQU C400H

            LXI D,Y ;Load DE with C300H (address Y)
            LXI H,X ;Load HL with C200H (address X)
            LXI B,Z ;Load BC with C400H (address Z)

            STC
            CMC     ;;Clear Carry flag

REP:   LDAX D ;Load A from memory pointed by DE
       SBB M  ;Subtract with borrow memory contents pointed by HL
       STAX B ;Store result in memory pointed by BC

       INX H
       INX D
```



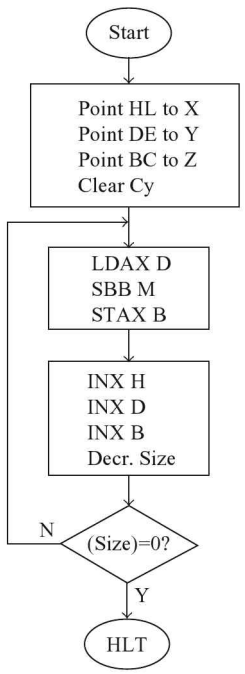


Fig. 14.8
Subtraction of two multi-byte numbers

```
INX B ;;;Increment the pointers HL, DE, and BC

LDA SIZE
DCR A
STA SIZE

JNZ REP ;If not zero jump to REP

HLT
```

■ 14.9 MULTIPLY TWO NUMBERS OF SIZE 8 BITS

Write an 8085 assembly language program to multiply two 8-bit numbers stored at locations X and Y. Store the 16-bit result in locations Z and Z+1. Also display the result in the address field.

Flowchart for the program

Flowchart for solving the problem is shown in Fig. 14.9.

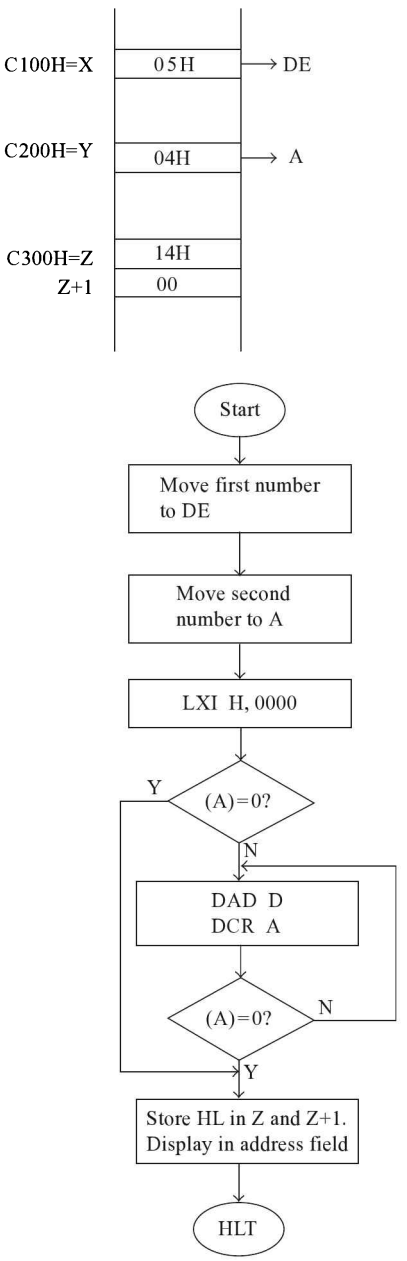


Fig. 14.9
Multiplication of two 8-bit numbers

Program to multiply two 8-bit numbers

```
;FILE NAME C:\ALS\MULT.ASM

;8085 ALP TO MULTIPLY 2 ONE BYTE BINARY NUMBERS STORED AT LOCATIONS X AND
;Y. DISPLAY THE 16 BIT RESULT IN THE ADDRESS FIELD.

;IT USES REPEATED ADDITION, AND AS SUCH IS NOT AN EFFICIENT METHOD.

            ORG C100H
  X:        DB 05H

            ORG C200H
  Y:        DB 04H

            ORG C000H
  Z:        EQU C300H
  CURAD:    EQU FFF7H
  UPDAD:    EQU 06BCH

            LXI H, X
            MOV E,M
            MVI D,00H ;;;Load DE with the byte at location X

            LXI H,Y
            MOV A,M   ;;Load A with the number at location Y

            LXI H,0000H;Initialise HL with 0000H

            CPI 00H
            JZ EXIT   ;;If A value is 00H jump to EXIT to display 0000

  AGAIN:    DAD D      ;Add HL and DE contents
            DCR A      ;Decrement A
            JNZ AGAIN  ;If result of decrement is not zero jump to AGAIN

      ;When we are out of this loop, the product will be in HL

  EXIT:     SHLD Z
            SHLD CURAD ;;HL contents stored in word locations Z and CURAD
            CALL UPDAD ;Display result in the address field

            HLT        ;Make sure to terminate with HLT instead of RST 1
```

14.9.1 Trace of the Program

In a program, knowing the actual values in affected registers and/or memory locations at the end of the execution of every instruction for a given set of sample data is called 'tracing the program'. Many times, it is easier to understand the working of the program by the trace for the program than by its flowchart or algorithm.

What follows is the trace for multiplication of two 8-bit numbers. The numbers are taken to be 05H and 04H.

Suppose several instructions are executed in a loop, then we indicate the contents of the affected register or memory location for each pass through the loop on a single line separated by '|' character.

For example, in the portion of the trace shown below we go through the loop four times. The values of HL and A at the end of each pass are shown separated by '|'. In the third line of the trace portion, J and NJ stand for 'jump' and 'no jump', respectively.

```
AGAIN: DAD D       ;HL = 0005H| 000AH| 000FH| 0014H
       DCR A       ;A  =   03H| 02H  | 01H  | 00H
       JNZ AGAIN   ;        J |   J  |   J  |  NJ
```

This convention results in easy tracing of the program as shown below.

```
;FILE NAME C:\ALS\MULT.ASM

;8085 ALP TO MULTIPLY 2 ONE BYTE BINARY NUMBERS STORED AT LOCATIONS X
;AND Y. DISPLAY THE 16 BIT RESULT IN THE ADDRESS FIELD.

;IT USES REPEATED ADDITION, AND AS SUCH IS NOT AN EFFICIENT METHOD.

           ORG C100H
X:         DB 05H

           ORG C200H
Y:         DB 04H

           ORG C000H
Z:         EQU C300H
CURAD:     EQU FFF7H
UPDAD:     EQU 06BCH

           LXI H, X     ; HL = C100H
           MOV E, M     ; E  = 05H
           MVI D, 00H   ; DE = 0005H

           LXI H, Y     ; HL = C200H
           MOV A, M     ; A  = 04H

           LXI H, 0000H; HL = 0000H

           CPI 00H      ; 04H vs 00H
           JZ EXIT      ; NJ

 AGAIN:            DAD D       ;HL = 0005H | 000AH | 000FH | 0014H
                   DCR A       ;A  = 03H   | 02H   | 01H   | 00H
                   JNZ AGAIN   ;       J   |   J   |   J   |  NJ

;When we are out of this loop, the product will be in HL

 EXIT: SHLD Z      ;(C300H) = 14H and (C301H) = 00H
       SHLD CURAD  ;(FFF7H) = 14H and (FFF8H) = 00H
       CALL UPDAD  ;Display 0014 in the address field

       HLT         ;Make sure to terminate with HLT instead of RST 1
```

■ 14.10 DIVIDE A 16-BIT NUMBER BY AN 8-BIT NUMBER

Write an 8085 assembly language program to divide a 16-bit number by an 8-bit number. The 16-bit number is at locations X and X+1. The 8-bit number is at location Y. Display quotient in the address field and remainder in the data field.

Flowchart for the program

Flowchart for solving the problem is shown in Fig. 14.10.

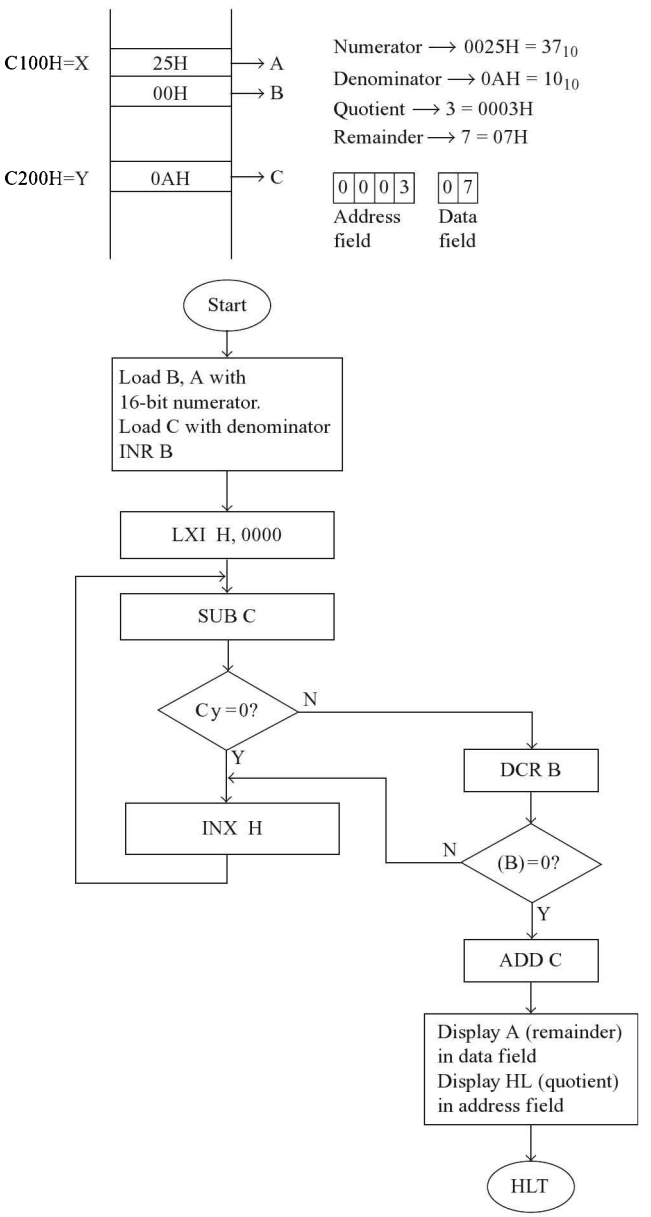


Fig. 14.10
Division of a 16-bit number by an 8-bit number

Program to divide a 16-bit number by an 8-bit number

```
;FILE NAME C:\ALS\DIV1.ASM

  ;8085 ALP TO DIVIDE A 16 BIT BINARY NUMBER STORED AT X AND X+1 BY AN 8
  ;BIT BINARY NUMBER STORED AT Y. DISPLAY QUOTIENT IN THE ADDRESS FIELD
  ;AND THE REMAINDER IN THE DATA FIELD.

;USES INEFFICIENT REPEATED SUBTRACT ALGORITHM

            ORG C100H
  X:        DW 0025H

            ORG C200H
  Y:        DB 0AH

            ORG C000H
  UPDAD:    EQU 06BCH
  UPDDT:    EQU 06D3H
  CURAD:    EQU FFF7H
  CURDT:    EQU FFF9H

      LXI H, X
      MOV A, M     ;;Load A with LS byte of the numerator.
      INX H
      MOV B, M
      INR B     ;;;Load B with one more than MS byte of numerator.

      LXI H,Y
      MOV C, M  ;;Load C with the denominator.

      LXI H,0;Initialize HL with 0. HL will finally have the quotient.

  AGAIN:

      SUB C         ;Subtract C contents from LS byte of numerator.
      JNC INC_QUO   ;If Cy = 0, jump to INC_QUO.
      DCR B         ;If Cy = 1, decrement MS byte of numerator.
      JZ EXIT       ;If zero, jump to EXIT. At this point, if we add
                    ;C contents to A, we get correct remainder in A.

  INC_QUO:
      INX H
      JMP AGAIN     ;;Increment quotient and jump to AGAIN.

  EXIT:
      ADD C         ;Now, A contains the remainder.
      STA CURDT     ;Store remainder in CURDT.

      SHLD CURAD
      CALL UPDAD    ;;Display quotient in address field.
      CALL UPDDT    ;Display remainder in data field.
      HLT
```

UPDDT routine: In this program, we have used the monitor routine UPDDT, which is used to update the data field on the kit. The routine displays the contents of the byte at symbolic location CURDT.

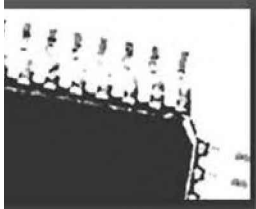
On the ALS kit UPDDT routine is at address 06D3H, and CURDT is byte location FFF9H. Thus, if we have 78H in location FFF97H, execution of 'CALL UPDDT' instruction results in display of '78' in the data field. CALL UPDDT is coded as CD D3 06.

In addition, if B register content is 00H, there will not be any dot at the end of the data field. If B content is 01H, a dot will be displayed at the end of the data field.

The contents of the registers A, B, C, D, E, H, L, and flags are all altered on executing CALL UPDDT.

The other common routine, RDKBD, is described in a later chapter.

1. Modify the program to exchange 10 bytes such that it exchanges 4 bytes, and provide trace for the program.
2. Write an 8085 assembly language program to find the largest and smallest numbers in a set of N byte-sized numbers. Display them in the address field.
3. Provide trace for the previous program when $N = 4$.
4. Write an 8085 assembly language program to find the number of 1s and 0s in a given byte. Display the number of 1s and 0s in the address field.
5. Write an 8085 assembly language program to check if a given set of N bytes are unsorted, sorted in ascending order, or sorted in descending order. Store in location RES the value 00, 01, or 02 accordingly.
6. Write an 8085 assembly language program to find if a triangle can be formed given the length of three sides. If triangle can be formed, store 01 in location RES, else store 00.
7. Write an 8085 assembly language program to find the length of a rectangle, given the breadth and the perimeter. Strore the result in location LEN.



15 Use of PC in Writing and Executing 8085 Programs

- Steps needed to run an assembly language program
  - *Assembly language program*
- Creation of .ASM file using a text editor
- Generation of .OBJ file using a cross-assembler
  - *Translation in prompt mode*
  - *Translation in command mode*
- Generation of .HEX file using a linker
  - *Prompt mode*
  - *Command mode*
  - *Data file mode*
- Downloading the machine code to the kit
  - *Setting up the ALS-SDA-85 kit for serial mode*
- Running the downloaded program on the kit
- Running the program using the PC as a terminal
  - *Running the entire program in a single operation*
  - *Running the program in single-step mode*
- Questions

In Chap. 5, we have discussed the use of the kit in the keyboard mode. In this case, the user had to translate the assembly language program into machine code, and enter the program in hexadecimal using the keyboard. This translation is quite monotonous, especially when the program is quite long.

If we have a computer with an ASCII keyboard, the program can be entered into the computer memory straightaway as an assembly language program. Then, the assembler program in the computer can translate this assembly language program to machine code, relieving the user from this monotonous task.

The machine code can finally be downloaded from the computer to the kit using RS-232C serial link. The user can then execute the downloaded program on the kit.

Also, the user can upload a machine language program from the memory of the kit to the hard disk or the floppy disk on the computer. The hard disk or floppy disk is non-volatile, which means it does

not lose the information even after power is switched off and then turned on. Thus the user programs can be stored as disk files on the computer, for downloading later to the kit for execution.

However, the cost goes up in serial mode, as a computer is a must in addition to the kit. Also, there must be an assembler software on the computer. The circuitry and the monitor software on the kit also is increased. But, with the reduction in cost of hardware and software, and the increase in cost of technical manpower, software development tools become necessary. Assembler is one such major software development tool. Use of assembler is explained in this chapter.

■ 15.1 STEPS NEEDED TO RUN AN ASSEMBLY LANGUAGE PROGRAM

The following steps are needed for running an assembly language program.

- Entering the program on a PC using a text editor. The file so created is given a suitable file name with .ASM extension.
- The file with .ASM extension is assembled to machine code using a 8085 cross-assembler. This generates the file with .OBJ extension.
- The file with .OBJ extension is linked to generate Intel Hex file using a linker. This generates the file with .HEX extension.
- The file with .HEX extension is downloaded to the kit using RS-232C serial interface and driver software.
- The program can now be executed on the kit using the keyboard mode of operation. Alternatively, serial monitor commands can be used to run the program. In such a case, the commands are issued using the PC, and the results are displayed on the CRT terminal of the PC. This method is more user-friendly, and the development of programs will be faster. Figure 15.1 illustrates the steps needed for running an assembly language program.

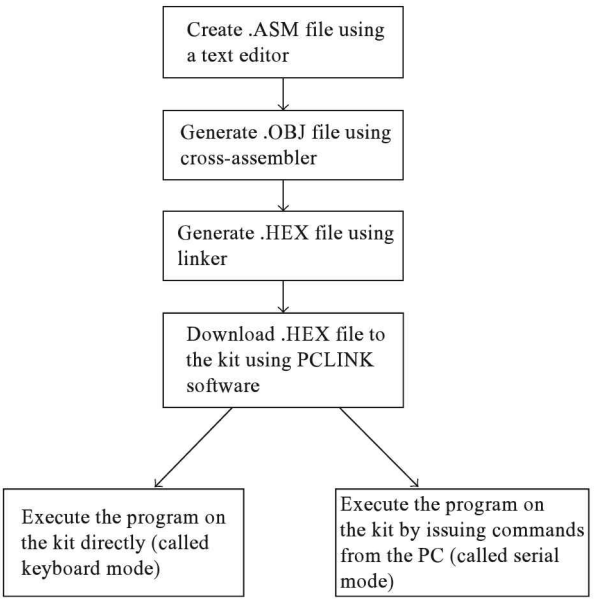


Fig. 15.1
Steps needed to run an assembly language program

To illustrate the various processes starting from entering the program, to final testing of the program, we consider a very simple assembly language program.

15.1.1 Assembly Language Program

Let us say, we have some 8-bit numbers in symbolic memory locations X and Y. We want to compute their product and store the 16-bit result in symbolic memory location Z and Z+1.

We need to have memory locations X and Y in the RAM area of the kit. The RAM space in the kit is C000H to C7FFH. So let us arbitrarily choose X as memory location C100H, Y as memory location C200H, and Z as memory location C300H. Let us say we want the program to start at location C000H.

Assembly language program to solve this simple problem is as follows. This problem was discussed towards the end of the previous chapter.

```
      ORG C100H    ; ORG can be replaced by ORIGIN
X:    DB 04H       ; DB can be replaced by .DB, BYTE, or .BYTE

      ORG C200H
Y:    DB   05H

      ORG C300H
Z:    DW           ;DW can be replaced by .DW, WORD, or .WORD

      ORG C000H
CURAD:EQU  FFF7H  ;EQU can be replaced by EQUAL
UPDAD:EQU  06BCH

BEGIN:LXI H,X
      MOV E, M
      MVI D, 00H

      LXI H, Y
      MOV A, M

      LXI H, 0000H
      CPI 00H
      JZ EXIT

AGAIN:DAD D
DCR A
JNZ AGAIN

EXIT: SHLD Z
      SHLD CURAD
      CALL UPDAD
      HLT          ;CHANGE THIS TO 'RST 1' FOR SERIAL MODE
      END BEGIN
```

Assembly language programs make use of the following features in addition to instruction mnemonics.

- Assembler directives like ORG, DB, DW, EQU, etc.
- Symbolic addresses like X in 'LXI H, X' instruction
- Labels like EXIT for an instruction like SHLD Z
- Comments are liberally used to make the program understandable.

Assembler Directives: Assembler directives are also called as pseudoinstructions. There are a number of assembler directives. They enable us to control the way a program is assembled and listed. They do not generate any machine code for execution. The most commonly used assembler directives are ORG, DB, DW, EQU, and END.

ORG Directive: This directive provides the origin or starting address for the assembly. It can also be written as ORIGIN. If this directive is not used, the starting address defaults to 0000H. For example, ORG C100H directive informs the assembler to assemble the code or data that follows the ORG directive from memory location C100H.

DB Directive: DB stands for define byte. It can also be written as .DB or .BYTE or BYTE. This directive reserves 1 byte of memory and initializes it with the value following the DB directive. If no value is indicated after the DB directive, the value defaults to 00H. If more than one value is indicated with comma as the separator, they are stored in consecutive byte locations.

Example 1

```
        ORG C150H
P:      DB 12H
```

This tells the assembler that P is a symbolic memory location where a byte value of 12H is stored. Because of the ORG C150H directive, symbolic address P is treated as memory address C150H. In the previous example, 12H could have been written in any of the following equivalent ways:

—12h (in hexadecimal notation);
—18 or 18D or 18d (in decimal notation);
—00010010B or 00010010b or 10010B or 10010b (in binary);
—22O or 22o or 22Q or 22q (in octal notation).

Example 2

```
        ORG C400H
Q:      DB
```

This tells the assembler that Q is a symbolic memory location where a byte value of 00H (default value) is stored. Because of the ORG C400H directive, symbolic address Q is treated as memory address C400H.

Example 3

```
        ORG C500H
R:      DB 04H,23,15D,00111010B,22Q,33O
```

Because of the ORG C500H directive, symbolic address R is treated as memory address C500H. Starting from location C500H the following values are stored in consecutive locations: 04H, 17H (23 decimal), 0FH (15 decimal), 3AH (00111010 binary), 12H (22 octal), and 1BH (33 octal).

Memory location containing 17H (23 decimal) is referred to as symbolic location R+1. Similarly, location R+2 contains 0FH, etc.

DW Directive: DW stands for define word. It can also be written as .DW or .WORD or WORD. This directive reserves one word of memory (2-byte locations) and initializes it with the value following the DW directive. If no value is indicated after the DW directive, the value defaults to 0000H. If more than one value is indicated with comma as the separator, they are stored in consecutive word locations.

Example 1

```
        ORG C150H
P:      DW 1234H
```

This tells the assembler that P is a symbolic memory location where a word value of 1234H is stored. Because of the ORG C150H directive, symbolic address P is treated as memory address C150H. In location P the value 34H is stored, and in location P+1 the value 12H is stored. This is the byte reversal method of word information storage. In the earlier example, 1234H could have been written using decimal, octal, or binary notation also.

Example 2

```
        ORG C400H
Q:      DW
```

This tells the assembler that Q is a symbolic memory location where a word value of 0000H (default value) is stored. Because of the ORG C400H directive, symbolic address Q is treated as memory address C400H.

Example 3

```
        ORG C500H
R:      DW 04H, 23
```

Because of the ORG C500H directive, symbolic address R is treated as memory address C500H. Value 0004H is stored in byte reversal form in word location C500H, and 0017H (23 decimal) is stored in byte reversal form in word location C502H. Memory location containing 17H (23 decimal) is referred to as symbolic location R+2.

EQU Directive: This directive equates the label to the left of EQU directive with the value to the right of the EQU directive. The value on the right of the EQU directive could be a number or an arithmetic expression or another symbol.

Normally, there will be a number to the right of the EQU directive. The EQU directive can also be written as EQUAL. The EQU directive does not reserve any memory locations, and it does not initialize the contents of any memory location.

Example

```
CURAD:    EQU FFF7H
```

This indicates that wherever we come across CURAD, it is to be treated as FFF7H. For example, if we come across the instruction 'SHLD CURAD', it is to be treated as 'SHLD FFF7H'.

END Directive: This directive defines the physical end of a program. There can be a symbolic address immediately following the END directive. It is optional. If the address is specified, it indicates the program starting address. Without this address, the program execution will be from the first executable instruction in the program.

In the earlier program we have used 'END BEGIN'. It could have been just END directive without the address following. In fact, the program is correctly translated even without the END directive. As such, END is not compulsory in a program. The assembly process is terminated when end of the file is encountered.

What is the need for END directive when HLT instruction informs the processor to halt? The answer is: HLT is an executable instruction of 8085. It is translated by the assembler. Generally HLT is the last instruction in many simple programs. But, in more lengthy programs, HLT instruction

can be seen at many places in the program. In such a case, the translation operation by the assembler should not stop at the first occurrence of the HLT instruction! The END directive appears only at the end of the program, and clearly indicates to the assembler that the translation is to be stopped.

Symbolic Addresses: In the earlier program we came across the instruction 'LXI H, Y'. In this instruction, Y is a symbolic memory location that stands for memory location C200H. Thus 'LXI H, Y' is same as 'LXI H, C200H'. This is because, using the ORG directive as shown below, we have indicated that Y stands for memory location C200H.

```
        ORG C200H
Y:      DB    05H
```

Similarly X stands for memory location C100H, and Z stands for memory location C300H.

Labels: In the earlier program we see the instruction 'AGAIN: DAD D'. This instruction is at memory location C012H. In this instruction, AGAIN is a label. It stands for memory location C012H, where the instruction is stored. There must be a colon between a label and an instruction.

A little later in the program we come across the instruction 'JNZ AGAIN', which now stands for 'JNZ C012H'. With this feature, the programmer simply provides a symbolic name or label to a branch destination, and uses that symbolic name in the branch instruction as the branch destination. Similarly in the instruction 'EXIT: SHLD Z', EXIT is a label, which stands for memory location C017H.

Comments: Comments are liberally used to make the program understandable. What is present in a line after a semicolon is treated as a comment, and is not translated. A comment can be just after an instruction or an assembler directive, and is separated by a semicolon. If we want the entire line to be a comment, the line should start with a semicolon.

■ 15.2 CREATION OF .ASM FILE USING A TEXT EDITOR

The assembly language program is entered using any editor, preferably a screen editor. While entering the program, the exact column at which label, instruction, and comment start is unimportant. But the instruction should start after column 1. However, the program is easy to read if the various fields are properly aligned. Any editor generally provides a help feature, using which the user can become familiar with the editor commands, and use them to enter the program.

The file so created is given a suitable file name with .ASM extension. For example, the earlier program may be entered as a file using, say, Norton editor with MULT.ASM as the file name. This .ASM file is called the *source file*.

■ 15.3 GENERATION OF .OBJ FILE USING A CROSS-ASSEMBLER

The MULT.ASM created using an editor is just a text file. It cannot be directly executed. We have to first assemble, and then link it. The assembly step involves translating the assembly language program into machine code. This step generates an .OBJ file.

In the examples given in this text, we have used '2500 A.D. 8085 cross-assembler—version 4.01'.

15.3.1 Translation in Prompt Mode

Type 'X8085<cr>' to perform translation in prompt mode. Here, '<cr>' stands for 'carriage return', which is done by typing the key on which 'Enter', is marked. The assembler prompts at every step in this mode. The advantage then is that there is no need to memorize the required steps.

After typing 'X8085<cr>' the assembler will prompt with:

Listing destination? (N, T, P, D, E, L, <cr> = N>)

where listing stands for listing the source code, object code, and details of errors encountered, if any. The listing could be on terminal, or printed using printer, or stored as a file with .LST extension on disk. Further the listing, if desired, could be confined to listing only erroneous source code lines.

In the equation shown, the abbreviations stand for:

N—None (no listing will be made),
T—Terminal (listing will be on terminal),
P—Printer (listing will be printed using printer),
D—Disk (listing stored as a .LST disk file),
E—Error only will be listed,
L—List On/Off (refer to manual for details).

If we are not interested in listing, just type '<cr>'. However, it is good to respond with typing 'T'. Then, the listing will be on terminal. The listing will pause at the source code where an error occurs. The user can note the error, and then type '<cr>' to proceed further and see the next error. If we respond with typing 'E', the assembler will prompt the user as follows:

Error only listing destination? (T, P, D, <cr> = T)

If we respond with '<cr>', the errors only will be listed on the terminal. In case many errors are encountered during the assembly process, it is better to generate a .LST file on the disk by responding to the prompt with 'D'. Anyway, after the user responds to the prompt for listing destination, the assembler prompts the user for the source file as shown:

Input filename :

The user should respond with the source file name. In this example, the user should respond with 'MULT.ASM'. Even 'MULT' will suffice. Then the assembler prompts the user for output file name as follows:

Output filename :

If the user responds with '<cr>', the output file name in this example would be 'MULT.OBJ'. If the user responds with 'DIV', the output file name would be 'DIV.OBJ'. Generally, the user responds to the output file name prompt with '<cr>'.

Suppose we choose to generate MULT.LST file on disk for this example. Then using a text editor we can see the contents of MULT.LST and identify the errors. After this, we make the necessary corrections in MULT.ASM file using the editor.

Finally, when the assembly process is successful without any error, the result will be placed in MULT.OBJ file.

15.3.2 Translation in Command Mode

The translation in command mode takes less number of steps. But the steps have to be remembered. We generally use this mode when we are very familiar with the necessary steps.

Type 'X8085 MULT.ASM <cr>' to perform translation in command mode. Even 'MULT' will suffice in place of 'MULT.ASM'. The source code in MULT.ASM is translated to the object code. The result will be placed in MULT.OBJ file.

To help in debugging the source code during assembly, we commonly use the following options depending on the user taste and convenience. It is assumed that the source file name is 'MULT.ASM'.

Option	*Function*
X8085 MULT-T	Displays listing on terminal. It pauses when error is encountered Continues when <cr> is pressed
X8085 MULT-P	Prints listing using printer
X8085 MULT-D	Generates .LST file on disk
X8085 MULT-ET	Displays only error listing on terminal
X8085 MULT-EP	Prints only error listing using printer
X8085 MULT-ED	Generates .LST file containing errors only

Suppose we choose to generate MULT.LST file on disk for this example. Then using a text editor we can see the contents of MULT.LST and identify the errors. After this, using the editor we make the necessary corrections in MULT.ASM file.

Finally, when the assembly process is successful without any error, the result will be placed in MULT.OBJ file. For more details about the assembler, refer to the assembler manual.

■ 15.4 GENERATION OF .HEX FILE USING A LINKER

The next step is to link the object module MULT.OBJ, which contains the machine code. We use the '2500 A.D. linker'. The linker enables the user to write assembly language programs consisting of several object modules. The linker will resolve external references and performs address relocation. This linker is capable of generating all of the most common file formats. Thus, it eliminates the need for an additional format conversion. The linker may be invoked in prompt mode, command line mode, or data file mode.

15.4.1 Prompt Mode

To run the linker in this mode type 'LINK85<cr>'. The linker will respond with a prompt requesting the user for file name. In this example, respond with 'MULT.OBJ<cr>'. Even 'MULT<cr>' will suffice.

Then the linker will prompt for the offset address. This offset value input by the user is added to the value of any ORG statements in the file. Generally, the user responds with <cr>, so that there is no offset value.

The linker then prompts for an output file name. Generally the user responds with <cr>. This causes the linker to generate an output file with the same name as the input file, but with a three-character extension that is determined by the output file type.

After the output file name is entered (or after responding with <cr>), the linker will prompt for library file names. We respond with <cr> in simple programs where we are not using library file names.

Then the linker will prompt for any linker options. For this we respond with <cr>, which results in generating .HEX file. In our example, the file created by the linker will be MULT.HEX, which will be in Intel HEX format. For more details about the linker refer to the 'X8085 cross-assembler' manual.

15.4.2 Command Mode

To run the linker in this mode type 'LINK85 -C MULT.OBJ<cr>'. Just 'MULT' is enough in place of 'MULT.OBJ' in the above command. The -C option indicates that we are running the linker in command mode. This command generates MULT.HEX file, which will be in Intel Hex format. For more details about the possible options, refer to 'X8085 cross-assembler' manual.

15.4.3 Data File Mode

Refer to 'X8085 cross-assembler' manual for details.

Intel Hex format: MULT.HEX file is shown below.

```
:01 C100 00 04 3A
:01 C200 00 05 38
:02 C300 00 00 00 3B
:10 C000 00 21 00 C1 5E 16 00 21 00 C2 7E 21 00 00 FE 00 CA 90
:10 C010 00 17 C0 19 3D C2 12 C0 22 F7 FF 22 00 C3 CD BC 06 D3
:01 C020 00 76 A9
:00 0000 01 FF
```

The file consists of several lines called records. A record starts with the character ':'. The next two characters indicate the record length field in hexadecimal. If this value is 00, as it is in the last record, it indicates EOF (end-of-file) record. This will be the last line of the file. The next four characters indicate the load address field in hexadecimal. The next two characters indicate the record-type field. It will be 00 for a data record, and 01 for end of the file record. Even program code is stored as data record. Thus, only the last record will have 01 in this field. After the record-type field, except the last two characters, we have the data bytes. The last two characters form the check sum. It is generated as the 2's complement of modulo 8-bit addition of length field, load address field, record-type field, and data bytes.

For example, in the fourth line of MULT.HEX we have

```
:10 C000 00 21 00 C1 5E 16 00 21 00 C2 7E 21 00 00 FE 00 CA 90
```

It is interpreted as follows. 10 indicates that there are 10H = 16 data bytes in the record. C000 indicates that they are stored in locations starting from C000H. The 00 indicates that the record-type is Data record. The 16 data bytes in hexadecimal are 21,00, C1, ... , and CA. 90 is the check sum and is obtained as follows.

10 + C0 + 00 + 00 + 21 + 00 + C1 +⋯+ CA = 70 with carry of 5. Ignoring the carry we get the result of modulo 8-bit addition. Thus the result of modulo 8-bit addition is 70H = 0111 0000B. Its 2's complement is 1001 0000B = 90H, which is the check sum.

■ 15.5 DOWNLOADING THE MACHINE CODE TO THE KIT

Now we have to download the MULT.HEX file, which is in Intel Hex format to the 8085 kit. A driver program that is used for communication between a PC and the ALS-SDA-85 kit using RS-232C interface does this. In the discussion that follows, we use PCLINK as the driver software for communication. The kit is assumed to be ALS-SDA-85M. PCLINK driver program can be used with any 8085 kit that has RS-232C interface.

First of all connect the kit to COM2 serial port of the PC using RS-232C cable. This is because generally in present-day computers, COM1 port is used for mouse, and is not free as such. However, if COM1 port is free, it can be used to connect to the kit. In the following discussion, it is assumed that COM2 port is used for connecting the kit.

Invoke the driver program by typing 'PCLINK<cr>'. It is a menu driven program. The PCLINK program first displays a welcome message, and then displays the main menu as shown in the following.

MAIN MENU

1 Terminal mode
2 Disk catalog
3 File download
4 File upload
5 Configuration
6 Exit program

Enter your choice [1–6]?

First of all, select configuration choice by typing 5. Then the configuration menu appears on the screen. It indicates the current status, and allows you to change the current status by displaying the menu as follows.

CONFIGURATION MENU

1 Baud rate
2 Word length
3 Stop bits
4 Parity
5 Serial port
6 Exit Configuration

Enter your choice [1–6]?

Select serial port choice by typing 5. Then the display indicates the current status, and allows you to select COM1 or COM2 by displaying the menu as follows.

SERIAL PORT

1 COM1
2 COM2

Enter your choice [1–2]?

Select COM2 by typing 2. Then the display indicates the current status. The current status of COM2 will be generally as follows.

Baud rate : 9600 Word length : 8 bits Parity : none
Stop bits : 1 Port : COM2

The menu allows you to change the current status by displaying the menu as follows.

To change, select from the following:

1 Baud rate
2 Word length
3 Stop bits
4 Parity
5 Serial port
6 Exit Configuration

Enter your choice [1–6]?

The current status with baud rate of 9,600, word length of 8 bits, no parity bit, and 1-stop bit is quite satisfactory. So we select Exit configuration choice by typing 6. Then once again the Main menu as shown in the following is displayed.

MAIN MENU

1 Terminal mode
2 Disk catalog
3 File download
4 File upload
5 Configuration
6 Exit program

Enter your choice [1–6]?

Select terminal mode choice by typing 1. This option allows us to use the PC as an ordinary display terminal. Whatever we type on the PC keyboard will appear on the screen and will be sent to the kit. Whatever is transmitted by the kit will appear on the PC screen. When we are in this mode, we have to press 'F10' key to exit the terminal mode and get back to Main menu.

15.5.1 Setting up the ALS-SDA-85 Kit for Serial Mode

When COM2 is programmed for 9,600 baud, it is necessary that the kit is also programmed for 9,600 baud. Loading location FFA6H on the kit with 0AH and location FFA7H with 00H does this. As per the kit manual, this sets the baud rate to 9,600. Then the 'Reset' button on the kit is pressed. Finally, the keys 'E' and '0' on the kit are pressed, which transfers the control to the PC. The keyboard on the kit will now be disabled (except the 'Reset' and 'Vect Intr' key). The '>' prompt will now appear on the PC screen. Now we are ready to use the 'ALS-SDA-85 serial monitor version 2.00' commands. For setting up 8085 kits from other manufacturers, you have to refer to relevant kit manuals.

To know the various commands available in the ALS serial monitor, type 'H<cr>' when the '>' prompt appears on the screen. Remember to type 'H' and not 'h'. It accepts commands typed in capitals only. Now the following menu will be displayed on the PC screen.

**** ALS-SDA-85 Monitor Ver 2.00 ****

* Summary of Monitor Commands * Memory Commands * Utility Commands

Memory Commands	*Utility Commands*	** Execution Commands*	** Host Utilities*
D Display	X Examine Register	G Execute	U File Upload
M Modify	A Assemble	S Single Step	L File Download
B Block Move	Z Disassemble		
I Insert	E EPROM Programmer		
K Delete	R Cassette Save		
F Block Fill	P Cassette Load		
C Block Complement	H Help		

The commands available in the serial mode of other kits can be different to some extent. Refer to relevant kit manuals for details.

To download the MULT.HEX file from the PC to the kit, type 'L' at the '>' prompt. The system prompts the user to enter offset. Generally, we respond with <cr>. Then the following message will be displayed.

Go to main menu by pressing 'F10' key, and select option 3

So we press 'F10' key. Then the Main menu appears on the screen. Now, select option 3 (File download option). Then the following prompt appears on the screen.

Name of file to download?

Type 'MULT.HEX<cr>', or just 'MULT<cr>'. The download operation from the PC to the kit starts. When the download operation is over, the main menu appears on the screen.

If our interest is to run the program on the kit using keyboard mode, we respond with selecting option 6 (exit program). If we desire to run the program on the kit using serial mode, we select option 1 (terminal mode).

■ 15.6 RUNNING THE DOWNLOADED PROGRAM ON THE KIT

Now that the program has been downloaded to the kit from the PC, we can run the program in the keyboard mode, which is quite familiar by now. Just press the 'Reset' key on the kit. The '-Sda 85' prompt appears on the seven-segment display of the kit. Now we can run the program by typing 'Go' key followed by the starting address of the program. The result is displayed in the address or data field of the display, if we have used UPDAD or UPDDT utilities in the program. Otherwise, we can use 'subst mem' key followed by memory address to check the results stored in memory locations.

■ 15.7 RUNNING THE PROGRAM USING THE PC AS A TERMINAL

Now that the program has been downloaded to the kit from the PC, we can run the program on the kit by issuing commands to the kit in serial mode, using the PC as a terminal. The result is also transmitted from the kit to the PC for display on the terminal. In this mode, the six-digit display on the kit will be

blank. The program can be run in a single burst without any pause after every instruction execution. Or, we can step through the program one instruction at a time for debugging purposes.

15.7.1 Running the Entire Program in a Single Operation

We generally run the program in this mode and hope to get the correct results straightaway. If we are unsuccessful, we attempt single-stepping through the program.

At the '>' prompt, type 'G'. Notice the absence of <cr> after G. The system prompts as follows:

Starting address: xxxx - yy/

where xxxx is a memory address and yy is the content of that memory location. It allows the user to respond with the desired starting address. If our program is to be executed from location xxxx, we just respond with <cr>. If our program is from location C000H, we respond with 'C000<cr>'.

Then the program is executed in a fraction of a second, and the result 0014 is displayed on the terminal. In this case, 0014H (decimal 20) is the result of multiplication of 04H at C100H, and 05H at C200H.

If our program ends with a HLT instruction, as is the case, it appears that the system does not respond to commands any more, after the display of 0014. This is because the HLT instruction is executed by 8085, and so 8085 has entered the halt state. Thus the kit can no longer communicate with the PC. All we have to do is press 'Reset' button on the 8085 kit, and then press 'E' and '0' keys on the kit. The '>' prompt reappears, and we can issue commands from the PC keyboard.

In view of this, it is desirable to end our programs on the ALS-SDA-85 kit with 'RST 1' instruction instead of 'HLT' instruction. Even if our program uses UPDAD and UPDDT monitor routines, it is desirable to end with RST 1, when program commands are issued in serial mode. Then, as soon as 0014 is displayed on the terminal, the 8085 executes 'RST 1' instruction. This results in the control being transferred to the Monitor program in the EPROM of the kit. However, note that in the keyboard mode we must end the program with HLT instruction, if we are using UPDAD and/or UPDDT monitor routines.

15.7.2 Running the Program in Single-step Mode

This mode is useful when we find that our program has not yielded desired results after executing in a single burst. In other words, it is used for debugging the program.

At the '>' prompt, type 'S'. Notice the absence of <cr> after S. The system prompts as follows:

Starting address: xxxx - yy/

where xxxx is a memory address and yy is the content of that memory location. It allows the user to respond with the desired starting address for single-stepping. If our program is to be executed from location xxxx, we just respond with <sb>. Here <sb> stands for pressing the SPACE BAR key, which is the widest on the keyboard. If our program is from location C000H, we respond with 'C000<sb>'.

Then the system displays 'C000-21/', where 21 is the content of memory location C000H. Now we respond with <sb>. Then the instruction at C000H is executed, and it displays the address of the next instruction along with the contents of that location. Only when we respond with <sb> the instruction is executed, and the address of the next instruction and the contents of that location are displayed. If we desire to stop the single-stepping, and check the contents of various registers and memory locations, we respond with <cr> instead of <sb>. Then the '>' prompt reappears.

Examine registers command: To check register values, type 'X' without <cr>. The system prompts as follows:

Register :

The user is required to respond with the desired register name. If you want to see contents of the C register, type C without <cr>. Immediately, it responds with

C=xx-

where xx is the content of the C register, and it allows the user to enter a new value in place of xx. For example, if we want C register to have 36H, we respond with 36 and <cr> or <sb>. If the user does not want to change the contents of the register, he has to respond with <cr> or <sb>. If it is <cr>, the X command will be terminated, and the '>' prompt reappears. If it is <sb>, the system displays content of D register automatically, and allows the user to optionally enter a new value into D the register. Using <sb> repeatedly, we can see the contents, and if desired modify the contents, of all registers. If we start with Register A, the sequence of registers will be as follows:

A, B, C, D, E, F, I, H, L, SPH, SPL, PCH, and PCL

where F is the Flags register, SPH and SPL are the MS and LS bytes of SP, PCH and PCL are the MS and LS bytes of PC. The 8 bits of register I provide the interrupt mask status as shown in the following.

0	0	0	0	IE	M7.5	M6.5	M5.5

IE = 1 implies 8085 interrupts are enabled
IE = 0 implies 8085 interrupts are disabled

M7.5 = 1 implies RST7.5 interrupt is masked
M7.5 = 0 implies RST7.5 interrupt is unmasked

M6.5 = 1 implies RST6.5 interrupt is masked
M6.5 = 0 implies RST6.5 interrupt is unmasked

M5.5 = 1 implies RST5.5 interrupt is masked
M5.5 = 0 implies RST5.5 interrupt is unmasked

Display memory command: To display memory contents, type 'D' without <cr>. The system prompts as follows:

Starting address :

The user is required to respond with the desired starting address for memory display. The user is required to provide the memory address in hexadecimal without the H suffix and <cr> (e.g. 'C200<cr>').

Then the system prompts the user for ending address. When the user provides the ending address, the contents of the desired memory locations are displayed on the terminal, and the '>' prompt reappears. Using this command we can only see the contents of a number of memory locations, but not alter them. To modify the contents of memory locations we have to use the 'modify memory command'.

Continuing with single step after checking registers/memory: Let us say, we are satisfied with the register contents and memory contents after single-stepping through the program by a few instructions. Then we want to continue with the remaining program in single-step mode.

At the '>' prompt, type 'S' without <cr>. The system prompts as follows:

Starting address: xxxx - yy/

where xxxx is a memory address and yy is the content of that memory location. Here xxxx is the correct instruction address where the single-stepping has to continue. So keep responding with <sb> as many times as desired to single step through the instructions. To terminate single-stepping respond with <cr>, and the '>' prompt reappears. Once again, the register contents and memory contents can be checked, and the single-stepping may be continued till the end of the program.

1. List the steps needed for executing an 8085 assembly language program.
2. What are assembler directives? Explain with examples the following assembler directives.
   a. ORG
   b. DB
   c. DW
   d. EQU
   e. END
3. Explain Intel Hex format of storing a file.
4. What are the advantages of using 8085 kit in serial mode compared with keyboard mode?

16 Additional Assembly Language Programs

- Search for a number using linear search
  - *Program to perform linear search*
- Find the smallest number
  - *Program to find the smallest number*
- Compute the HCF of two 8-bit numbers
  - *Program to find the HCF of two given bytes*
- Check for two out of five code
  - *Program to check for two out of five code*
- Convert ASCII to binary
  - *Program to convert ASCII to binary*
- Convert binary to ASCII
  - *Program to convert two-digit hex to two ASCII values*
- Convert BCD to binary
  - *Program to convert a two-digit BCD to binary—method 1*
  - *Program to convert a two-digit BCD to binary—method 2*
- Convert binary to BCD
  - *Program to convert an 8-bit binary to BCD—method 1*
  - *Program to convert an 8-bit binary to BCD—method 2*
  - *Program to convert an 8-bit binary to BCD—method 3*
- Check for palindrome
  - *Program to check for palindrome*
- Compute the LCM of two 8-bit numbers
  - *Program to compute LCM*
- Sort numbers using bubble sort
  - *Program to perform sorting using bubble sort*
- Sort numbers using selection sort
  - *Program to perform sorting using selection sort*

■ Simulate decimal up counter
• *Program to simulate decimal up counter*
• *Generation of time delay*
■ Simulate decimal down counter
• *Program to simulate decimal down counter*
■ Display alternately 00 and FF in the data field
• *Program to alternately display 00 and FF in the data field*
■ Simulate a real-time clock
• *Program to simulate a real-time clock*
■ Questions

In Chap. 14, a number of simple assembly language programs were discussed. Those programs were manually translated by the user, were loaded using the keyboard, and finally executed using the commands issued by the keyboard of the kit. Further, in the previous chapter, the use of assembler, and that of PC in executing a program in serial mode were discussed.

From now on, the user is encouraged to use the PC to enter his program, do the translation, download the program to the kit, and run the program using commands issued by the PC in serial mode. In this chapter we deal with the simple 8085 assembly language programs mentioned previously in the list, along with their flowcharts.

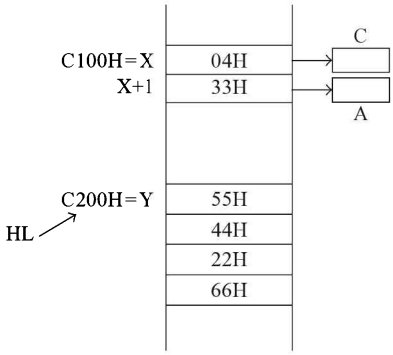
■ 16.1 SEARCH FOR A NUMBER USING LINEAR SEARCH

Write an 8085 assembly language program to search for a given byte in an array of bytes using linear search algorithm. Location X contains the size of the array and location X+1 contains the element to be searched. The elements of the array are stored from location Y onwards. The program should display in the address field, the search element and the position where it was found. If the search element is not found, the position should be indicated as 00.

Flowchart for solving the problem is shown in Fig. 16.1.

16.1.1 Program to Perform Linear Search

```
;FILE NAME C:\ALS\LINSRCH.ASM

  ;8085 ALP TO PERFORM LINEAR SEARCH. LOCATION X CONTAINS
  ;THE NUMBER OF BYTES TO SEARCH, LOCATION X+1 CONTAINS THE ELEMENT
  ;TO BE SEARCHED, AND LOCATION Y ONWARDS ARE THE ELEMENTS OF THE ARRAY.

  ;PROGRAM DISPLAYS IN THE ADDRESS FIELD, THE SEARCH ELEMENT AND THE
  ;POSITION WHERE IT WAS FOUND. IF THE SEARCH ELEMENT IS NOT FOUND,
  ;THE POSITION WILL BE INDICATED AS 00.

  ORG C100H
X: DB 04H,33H
```



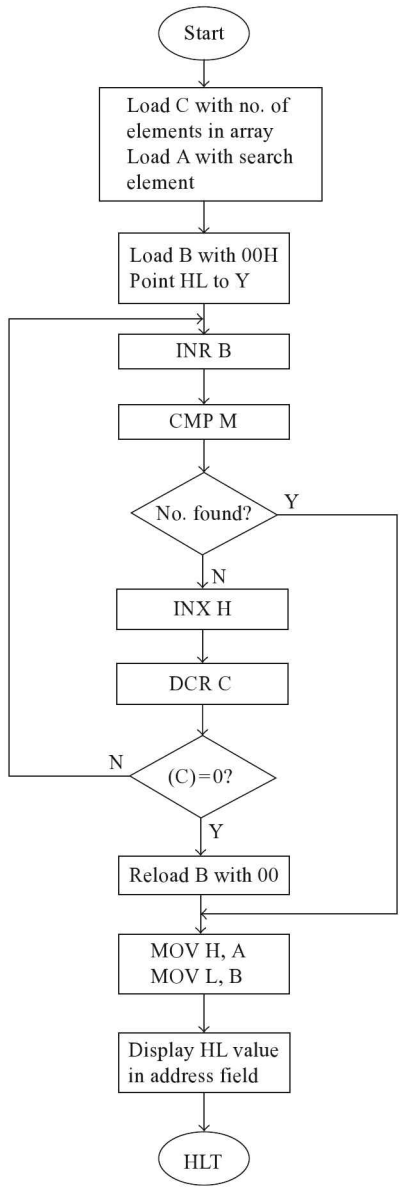


Fig. 16.1
Flowchart for linear search

```
        ORG C200H
Y:  DB 55H,44H,22H,66H
        ORG C000H
CURAD:  EQU FFF7H
UPDAD:  EQU 06BCH
        LXI H,X
        MOV C,M ;;Load C with the number of elements in the array.
        INX H
        MOV A,M ;;Load A with the element to be searched.
        MVI B,00H;Initialise B with 0.
        ;It finally indicates the position where the element was found.
        LXI H,Y ;Point HL to the beginning of the array.
REP:    INR B ;Increment B. B indicates element number being checked.
        CMP M ;Compare A and memory pointed by HL.
        JZ EXIT ;If they are same, jump to EXIT.
        INX H
        DCR C ;;Else, increment HL and decrement C.
        JNZ REP;If C value is nonzero, jump to REP.
        ;If we come out of this loop because C is 00,
        ;it means search is unsuccesful. Hence make B as 00.
        MVI B,00H ;Load B with 00H.
        ;At this point B will have the position where the element
        ;was found. If element was not found, B will be 00H.
EXIT:   MOV H,A    ;Load H with the search element.
        MOV L,B    ;Load L with the position the element was found.
        SHLD CURAD
        CALL UPDAD ;;Display the result in the address field.
        HLT        ;Make sure to end with HLT for keyboard mode.
                   ;For serial mode RST 1 is preferred.
```

■ 16.2 FIND THE SMALLEST NUMBER

Write an 8085 assembly language program to find the smallest of N 1-byte numbers. The N value is provided at location X, and the numbers are present from location X+1. Display the smallest number in the data field, and its location in the address field.

Flowchart for solving the problem is shown in Fig. 16.2.

16.2.1 PROGRAM TO FIND THE SMALLEST NUMBER

```
;FILE NAME C:\ALS\SMALL.ASM
  ;8085 ALP TO FIND THE SMALLEST OF N ONE BYTE NUMBERS. N VALUE
  ;IS STORED AT LOCATION X AND THE NUMBERS FROM LOCATION X+1.
  ;DISPLAY THE SMALLEST NUMBER IN THE DATA FIELD, AND ITS LOCATION
```

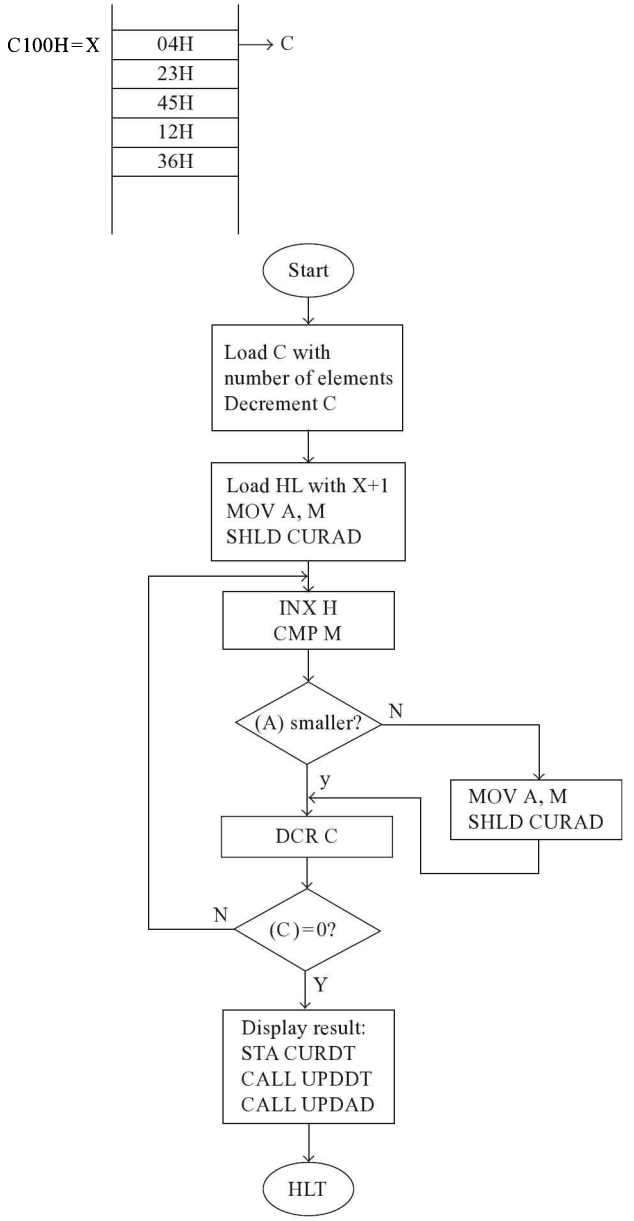


Fig. 16.2
Flowchart to find the smallest number

```
   ;IN THE ADDRESS FIELD.
    ORG C100H
X:  DB 04H,23H,45H,12H,36H
    ORG C000H
CURDT: EQU FFF9H
UPDDT: EQU 06D3H
CURAD: EQU FFF7H
```

```
UPDAD: EQU 06BCH

       LXI H, X
       MOV C,M     ;;Load C with the number of elements.
       DCR C        ;Decrement C. It indicates the number of comparisons
                    ;to be made to find the smallest element.

       INX H
       MOV A,M     ;;Load A with the first element.
       SHLD CURAD   ;Store its address in word location CURAD.

AGAIN: INX H        ;Increment the pointer HL.
       CMP M        ;Compare A with memory pointed by HL.
       JC SKIP    ;If A is smaller, jump to SKIP.

       MOV A, M  ;Else, load A with the smaller number pointed by HL
       SHLD CURAD;and store its address in word location CURAD.

SKIP:  DCR C     ;Decrement C.
       JNZ AGAIN ;If nonzero jump to AGAIN.

       ;Thus, when we come out of this loop, A will have the
       ;smallest element, and its address in word location CURAD.

       STA CURDT
       CALL UPDDT
       CALL UPDAD   ;;;Display the result in address and data field.
       HLT            ;Terminate with HLT only in keyboard mode.
                      ;For serial mode RST 1 is preferred.
```

■ 16.3 COMPUTE THE HCF OF TWO 8-BIT NUMBERS

Write an 8085 assembly language program to find the HCF of two 8-bit numbers. The numbers are stored at locations X and Y. Display the numbers in the address field, and their HCF in the data field.

Flowchart for solving the problem is shown in Fig. 16.3.

16.3.1 Program to Find the HCF of Two Given Bytes

```
;FILE NAME C:\ALS\HCF.ASM

       ;8085 ALP TO FIND THE HCF OF TWO SINGLE BYTE NUMBERS.
       ;THE NUMBERS ARE STORED AT LOCATIONS X AND Y. DISPLAY THE
       ;NUMBERS IN THE ADDRESS FIELD, AND THE HCF IN THE DATA FIELD.

       ORG C100H
X:     DB 0FH

       ORG C200H
Y:     DB 19H

       ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
```

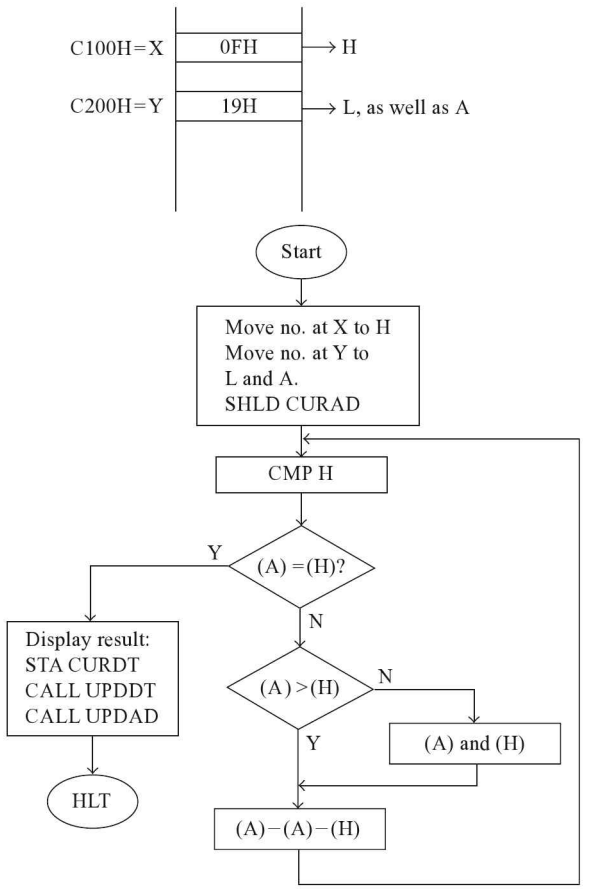


Fig. 16.3 Flowchart to find the HCF

```
CURAD: EQU FFF7H
UPDAD: EQU 06BCH

       LDA X
       MOV H, A ;;Load H with the number at location X.

       LDA Y
       MOV L, A ;;Load A and L with the number at location Y.
       SHLD CURAD;Store the two numbers in word location CURAD.

AGAIN: CMP H         ;Compare A and H contents.
       JZ EXIT       ;If they are same, jump to EXIT with HCF value in A.

       JNC DOWN_A    ;If A is larger, jump to DOWN_A, to reduce it.

       MOV L, A
       MOV A, H
       MOV H, L      ;;;Else, i.e. if A is smaller, exchange A and H.

DOWN_A: SUB H        ;Reduce A value by H contents.
        JMP AGAIN    ;Jump to AGAIN.

        ;When we come out of this loop, HCF value will be in A.
```

```
EXIT:  STA CURDT
       CALL UPDDT    ;;Display the HCF in data field.
       CALL UPDAD     ;Display the two numbers in address field.
       HLT            ;Use HLT only in keyboard mode,
                     ;and preferably RST 1 in serial mode.
```

■ 16.4 CHECK FOR '2 OUT OF 5' CODE

Write an 8085 assembly language program to check if the byte at location X belongs to '2 out of 5' code or not. It belongs to '2 out of 5' code, if the MS 3 bits are 000, and there are two 1s in the LS 5 bits. Display in the address field, the number and FF in the case of valid code, or the number and 00 in the case of invalid code.

Flowchart for solving the problem is shown in Fig. 16.4.

16.4.1 PROGRAM TO CHECK FOR '2 OUT OF 5' CODE

```
;FILE NAME C:\ALS\CHKCODE.ASM

       ;8085 ALP TO CHECK IF THE BYTE AT X BELONGS TO '2 OUT OF 5' CODE OR NOT.
       ;( IT IS VALID CODE IF THE MS 3 BITS ARE 000, AND THERE ARE TWO 1'S IN
       ;THE LS 5 BITS )

       ;PROGRAM DISPLAYS IN THE ADDRESS FIELD, THE NUMER AND FF
       ;IF IT IS VALID CODE, ELSE DISPLAYS THE NUMBER AND 00

       ORG C100H
X:     DB 14H

       ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH

       LDA X
       MOV H,A       ;;Load A and H with the number to be tested.

       MVI L, 00H    ;Load L with 00. It will be changed later to FF
                     ;if the number belongs to '2 out of 5' code.

       ANI 11100000B;Reset to 0 the LS 5 bits of A,
                     ;with MS 3 bits unchanged.

       JNZ ZERO      ;If MS 3 bits are not 000 jump to ZERO to indicate
                     ;that the number does not belong to '2 out of 5' code

       MOV A, H      ;Load A with the number to be tested, present in H.

       MVI C, 05H    ;Load C with 5, the number of bits to test.
       MVI D, 00H    ;Initialise D with 00. It will finally have
                     ;the number of 1's in the LS 5 bits.

LOOP:  RRC           ;Rotate right Accumulator A.
       JNC NOINCD    ;If there is no Carry, jump to NOINCD,
```

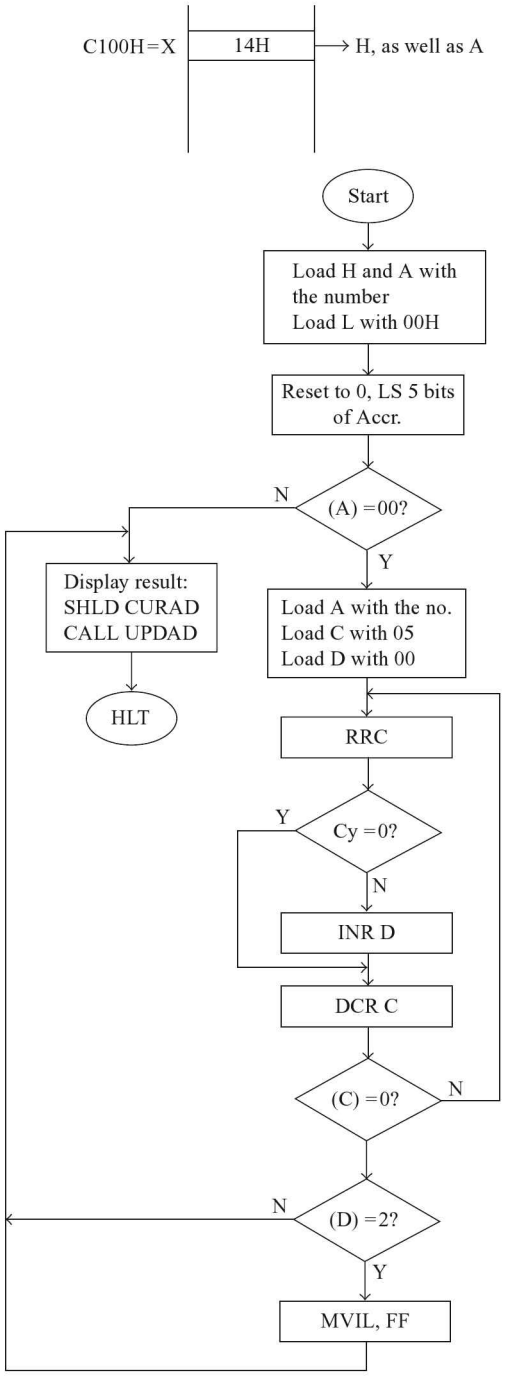


Fig. 16.4
Flowchart to check for two out of five code

```
        INR D           ;else, increment D.
NOINCD: DCR C           ;Decrement the counter C.
        JNZ LOOP        ;If not zero, jump to LOOP.
        ;When we come out of this loop, D will have
```

```
        ;number of 1's in the LS 5 bits.

        MVI A,02H
        CMP D ;;Compare D with 2.
        JNZ ZERO ;If D is not equal to 2, jump to ZERO.

        MVI L, FFH ;If D is equal to 2, load L with FFH.

        ;At this point H will have the number, and L will be FF or 00
        ;depending on whether the number belongs to '2 out of 5' code or not.

ZERO:   SHLD CURAD
        CALL UPDAD    ;;Display result in address field.
        HLT           ;Teminate with HLT only for keyboard mode,
                      ;and preferably with RST 1 for serial mode.
```

■ 16.5 CONVERT ASCII TO BINARY

Write an 8085 assembly language program to convert an ASCII hex character to the equivalent binary. ASCII hex number is at location X. Display the ASCII hex number and its binary equivalent in the address field.

Flowchart for solving the problem is shown in Fig. 16.5.

16.5.1 PROGRAM TO CONVERT ASCII TO BINARY

```
;FILE NAME C:\ALS\ASC_BIN.ASM

  ;8085 ALP TO CONVERT AN ASCII HEX CHARACTER TO BINARY.
  ;ASCII HEX NUMBER IS AT LOCATION X. DISPLAY THE ASCII NUMBER
  ;AND ITS BINARY(HEX) EQUIVALENT IN THE ADDRESS FIELD.

  ;Some details about ASCII codes

  ;ASCII code for character '0' is 30H, for '1' it is 31H, etc
  ;and for '9' it is 39H. The ASCII code for 'A' is 41H, for 'B'
  ;it is 42H, etc and for character 'F' it is 46H.

  ;So, to convert ASCII to hexadecimal value, we have to subtract
  ;30H in case it is a character from '0' to '9', and subtract 37H
  ;in case it is a character from 'A' to 'F'.

  ;For example, 42H - 37H = 0BH, where 42H is ASCII code for 'B'.

  ;NOTE: We display Hexadecimal value, instead of binary because
  ;to display binary we need more number of 7 segment L.E.D.s than
  ;available on the kit. Also, display in binary is more difficult
  ;to comprehend.

        ORG C100H
  X:    DB 43H

        ORG C000H
  CURAD: EQU FFF7H
  UPDAD: EQU 06BCH
```

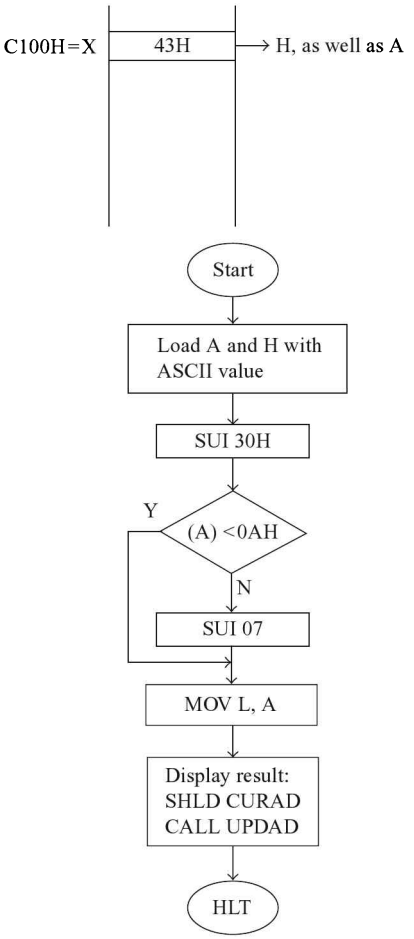


Fig. 16.5
Flowchart to convert ASCII hex to binary

```
        LDA X
        MOV H, A ;;Load A and H with contents of location X.

        SUI 30H ;Subtract 30H from A.
        CPI 0AH ;Compare result with 0AH.
        JC NUMB ;If A value is 00 to 09, jump to NUMB.
                ;Then, ASCII code corresponds to '0' to '9',

        SUI 07H ;else, subtract 07H again
                ;because ASCII code corresponds to 'A' to 'F'.

        ;At this point A has the hexadecimal number corresponding
        ;to the ASCII code in H.

NUMB:   MOV L, A

        SHLD CURAD
        CALL UPDAD;;;Display ASCII code and its hexadecimal
                   ;equivalent in the address field.
        HLT
```

■ 16.6 CONVERT BINARY TO ASCII

Write an 8085 assembly language program to convert a two-digit hexadecimal number to two equivalent ASCII codes. The two-digit hexadecimal number is at location X. Display the hexadecimal number in the data field and its equivalent ASCII code (2 bytes) in the address field.

Flowchart for solving the problem is shown in Fig. 16.6.

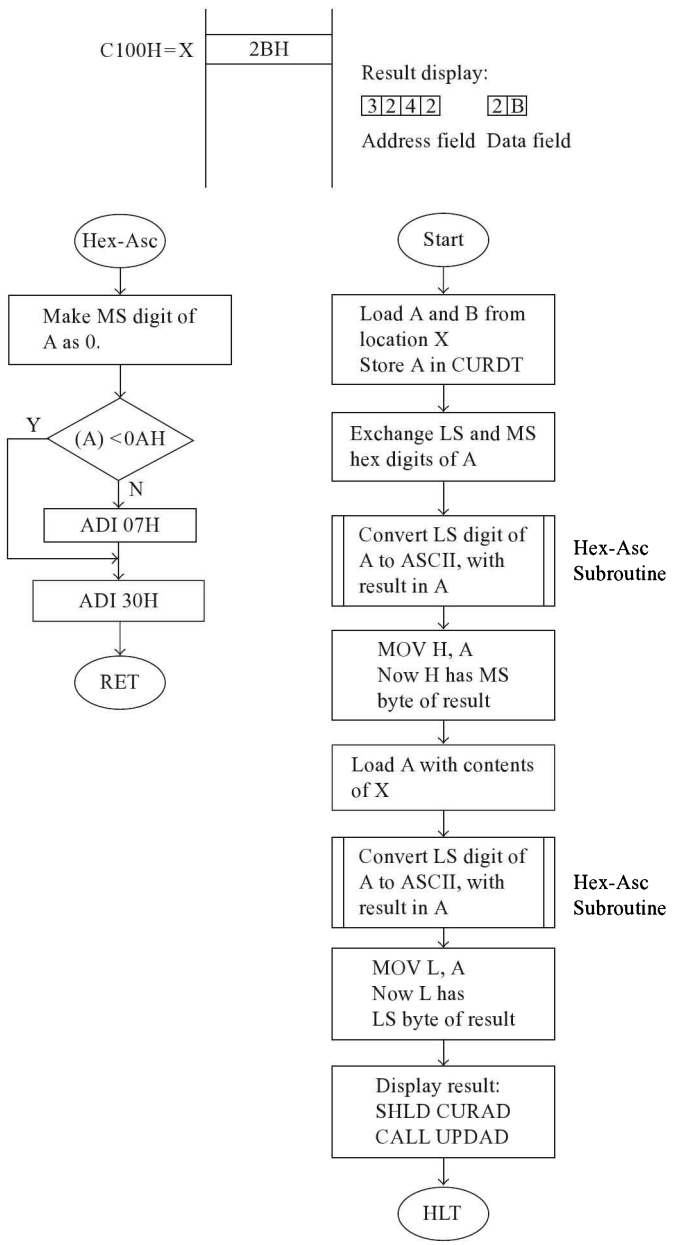


Fig. 16.6 Flowchart to convert two-digit hex to two ASCII bytes

16.6.1 Program to Convert Two-digit Hex to two ASCII Values

```
;FILE NAME C:\ALS\BIN_ASC.ASM

       ;8085 ALP TO CONVERT A 2 DIGIT HEXADECIMAL NUMBER TO
       ;EQUIVALENT TWO ASCII CODES. THE 2 DIGIT HEXADECIMAL NUMBER IS
       ;AT LOCATION X. DISPLAY THE HEXADECIMAL NUMBER IN THE DATA FIELD
       ;AND ITS EQUIVALENT ASCII CODE IN THE ADDRESS FIELD.

       ORG C100H
  X:   DB 2BH

       ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
CURAD: EQU FFF7H
UPDAD: EQU 06BCH

       LDA X
       STA CURDT
       MOV B, A ;;;Load A and B from location X. Store A in CURDT.

       RRC
       RRC
       RRC
       RRC ;;;;Rotate right A four times
             ;to exchange LS and MS digits of A.

       CALL HEX_ASC;Convert LS digit of the number in A to ASCII.
                   ;Result will be in A. Thus A will have ASCII
                   ;code of MS digit of number because of exchange.

       MOV H, A    ;Load H with ASCII code of MS digit.

       MOV A, B    ;Load A from B. So A will have original number.
       CALL HEX_ASC;Convert LS digit of the number in A to ASCII.
                   ;Result will be in A. Thus A will have ASCII
                   ;code of LS digit of number.

       MOV L, A   ;Load L with ASCII code of LS digit.

       SHLD CURAD
       CALL UPDAD ;;Display ASCII codes in address field.
       CALL UPDDT  ;Display the number in data field.
       HLT

       ;The subroutine shown below converts the LS hex digit in A, to its
       ;equivalent ASCII code. Result will be in A. To get the ASCII code we
       ;have to add 30H, if the character is in the range '0' to '9', and add
       ;37H if the character is in the range 'A' to 'F'.

HEX_ASC:
       ANI 0FH   ;Make MS 4 bits of A as 0000.
       CPI 0AH   ;Compare LS hex digit of A with 0AH.
       JC NUMB   ;If LS hex digit of A is 0 to 9, jump to NUMB
                 ;which adds 30H to get the ASCII code.

       ADI 07H   ;else add 07H, and then proceed to NUMB so that
                 ;effectively 37H is added to get ASCII code.

NUMB:  ADI 30H   ;Add 30H so that A will have the ASCII code.
       RET       ;Return to calling program
```

■ 16.7 CONVERT BCD TO BINARY

Write an 8085 assembly language program to convert a two-digit BCD number to binary. The two-digit BCD number is at location X. Display the BCD number and its binary equivalent in the address field.

Generally, there is more than one way of solving a given problem. To illustrate this idea, two methods of converting a BCD number to binary are described in this section. A flowchart for the first method of solving the problem is shown in Fig. 16.7a.

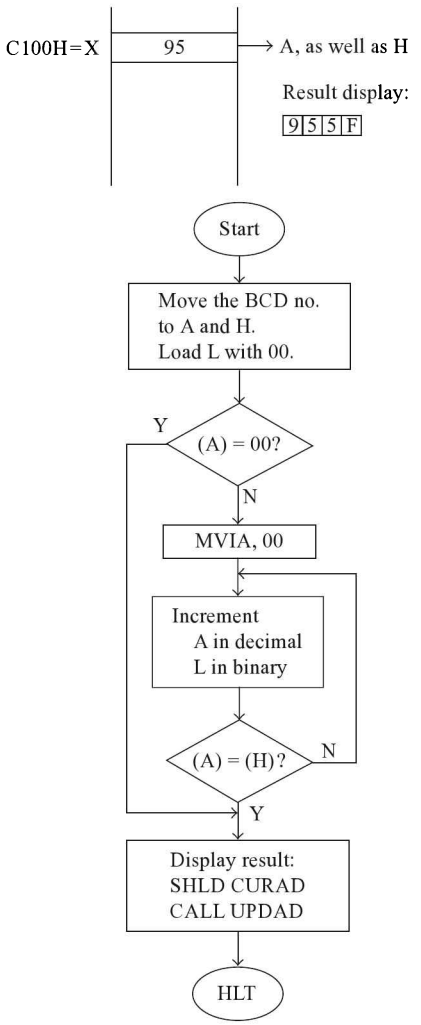


Fig. 16.7a
Flowchart to convert a two-digit BCD to binary — Method 1

16.7.1 Program to Convert a Two-digit BCD to Binary—Method 1

```
;FILE NAME C;\ALS\BCD_BIN.ASM

  ;8085 ALP TO CONVERT A 2 DIGIT BCD NUMBER TO BINARY.
  ;THE 2 DIGIT BCD NUMBER IS AT LOCATION X. DISPLAY THE BCD NUMBER
  ;AND ITS BINARY(HEX) EQUIVALENT IN THE ADDRESS FIELD.

  ;There are many ways of solving this problem. The method adopted
  ;here is as follows. Registers A and L are initialised to 00. Then
  ;A is incremented in decimal, and L is incremented in hexadecimal
  ;till A equals the given BCD value. Then L will have the equivalent
  ;hexadecimal value.

  ORG C100H
X: DB 95H

  ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH
       LDA X
       MOV H, A     ;;Load H and A with BCD number from location X.

       MVI L, 00H   ;Initialise L with 00H. Finally L will have
                    ;equivalent Hexadecimal number.
       CPI 00H      ;Compare the BCD number with 00.
       JZ EXIT      ;If the BCD number is 00, jump to EXIT
                    ;to result display in the address field.

       MVI A, 00H   ;Initialize A with 00.

REPEAT:
       ADI 01H
       DAA    ;;Increment A in decimal system.
       INR L  ;Increment L in binary(hex) system.
       CMP H  ;Compare A with the BCD number to be converted.

       CMC    ;As A will be less than H to start with, the
              ;comparison results in Cy becoming 1. So complement
              ;Cy flag to make it 0. Else, DAA gives wrong result.

       JNZ REPEAT;If A is not equal to BCD number, jump to REPEAT.

       ;When we come out of this loop, A and H will have the
       ;BCD number, and L will have equivalent hexadecimal number.

EXIT: SHLD CURAD
      CALL UPDAD ;;Display the result in address field
      HLT
```

Flowchart for the second method of solving the problem is shown in Fig. 16.7b.

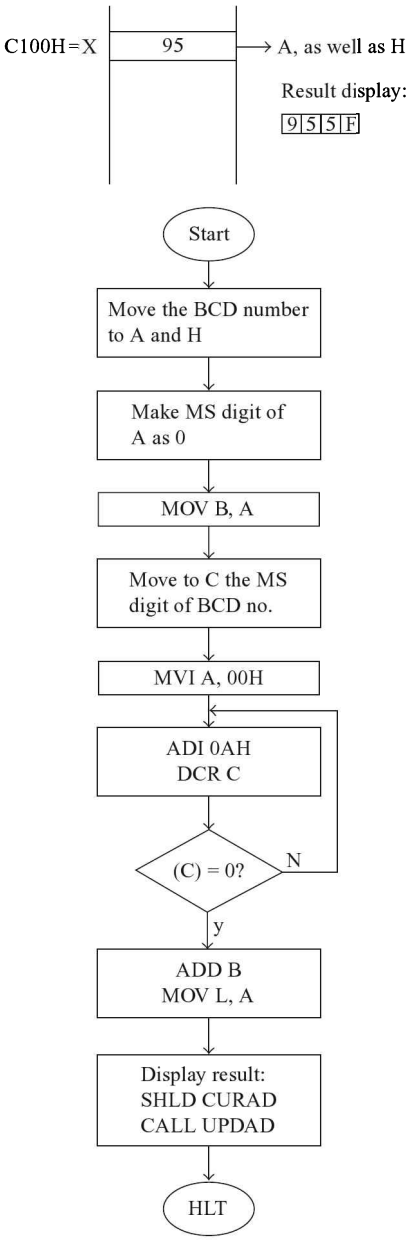


Fig. 16.7b
Flowchart to convert a two-digit BCD to Binary — Method 2

16.7.2 Program to Convert a Two-digit BCD to Binary—Method 2

```
;FILE NAME C:\ALS\BCDBIN2.ASM

        ;8085 ALP TO CONVERT A 2 DIGIT BCD NUMBER TO BINARY
        ;THE 2 DIGIT BCD NUMBER IS AT LOCATION X. DISPLAY THE BCD NUMBER
        ;AND ITS BINARY(HEX) EQUIVALENT IN THE ADDRESS FIELD.
```

```
        ;We use the following method for converting BCD to binary. Suppose
        ;we have the BCD number 45. This is 4 x 10 + 5. Or in hexadecimal,
        ;it is equivalent to 4 x 0AH + 5. Thus the method involves multi-
        ;plying by 0AH the MS digit of the number, and then adding the
        ;LS digit of the number.

        ORG C100H
 X:     DB 95H

        ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH

        LDA X
        MOV H, A;;Load A and H with the BCD number at location X.

        ANI 0FH  ;Make the MS 4 bits of A as 0000.
                 ;Thus A will now have the LS digit of the number.

        MOV B, A ;Load B with LS digit of the number.

        MOV A, H ;Reload A with the 2 digit BCD number in H.
        ANI F0H  ;Make the LS 4 bits of A as 0000.
        RRC
        RRC
        RRC
        RRC    ;;;;Exchange LS and MS digits of A.
                 ;Thus A will now have the MS digit of the number.

        MOV C, A ;Load C with MS digit of the number.

        ;The following 4 instructions multiply C contents with 0AH.

        MVI A, 00H   ;Initialise A with 00H.
AGAIN:  ADI 0AH      ;Add 0AH.
        DCR C        ;Decrement C.
        JNZ AGAIN    ;If not zero, jump to AGAIN.

        ;When we are out of this loop, A will have C contents x 0AH.

        ADD B    ;Add B contents. Now A will have equivalent Hex value.
        MOV L, A ;Load L with the result in A.

        SHLD CURAD

        CALL UPDAD ;;Display in address field, the BCD number
                   ;and its Hexadecimal value.
        HLT
```

■ 16.8 CONVERT BINARY TO BCD

Write an 8085 assembly language program to convert an 8-bit binary number to equivalent BCD number. The binary number is at location X. Display the binary number in the data field, and its equivalent BCD number in the address field.

There are various methods of solving this problem. Three of them are described in this text. A flowchart for the first method of solving the problem is shown in Fig. 16.8a.

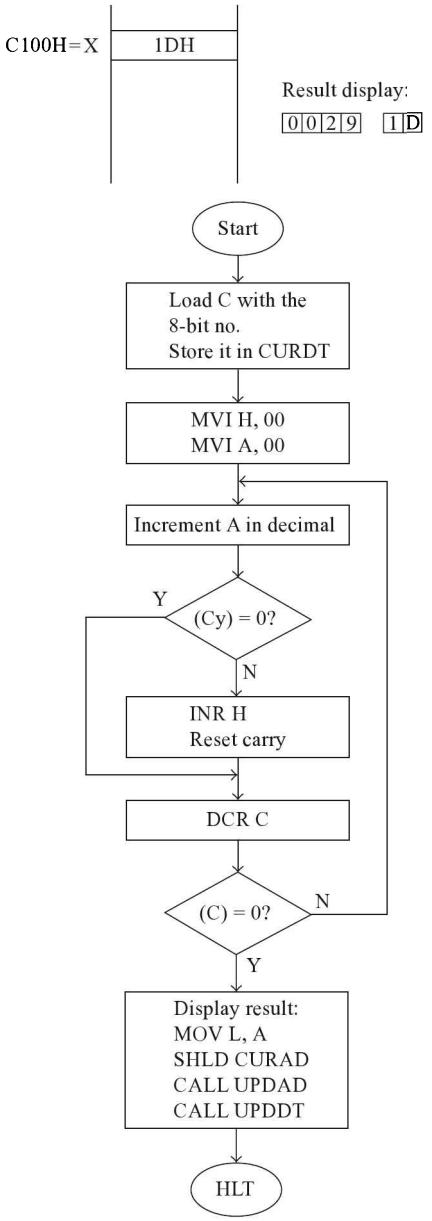


Fig. 16.8a
Flowchart to convert an 8-bit binary to BCD — Method 1

16.8.1 Program to Convert an 8-bit Binary to BCD—Method 1

```
;FILE NAME C:\ALS\BIN_BCD.ASM

  ;8085 ALP TO CONVERT A 8 BIT BINARY NUMBER TO
  ;EQUIVALENT BCD NUMBER. THE BINARY NUMBER IS AT LOCATION X.
  ;DISPLAY THE BINARY NUMBER IN THE DATA FIELD
  ;AND ITS EQUIVALENT BCD NUMBER IN THE ADDRESS FIELD.

  ;The method used in this program is as follows. Suppose we want to
  ;convert CF to equivalent BCD. We move this number to a register,
  ;say, C register. We load A with 00. Now we keep decrementing C in
  ;hexadecimal and incrementing A in decimal till C becomes 00. If
  ;there is a Carry during incrementing of A, we accumulate these
  ;carries in another register, say H. Finally, H and A will have
  ;the BCD equivalent.

  ORG C100H
X:DB 1DH

  ORG C000H

  CURDT:EQU FFF9H
  UPDDT:EQU 06D3H
  CURAD:EQU FFF7H
  UPDAD:EQU 06BCH

       LDA X
       STA CURDT
       MOV C, A ;;;Load C from location X. Store it in CURDT.

       MVI H, 00H ;Initialise H to 0. Finally H will have
                  ;MS byte of BCD result.

       MVI A, 00H ;Initialise A to 0. Finally A will have
                  ;LS byte of BCD result.

       ;The next 7 instructions decrement C in hexadecimal, and
       ;increment A in decimal till C becomes 00. During
       ;incrementing of A if there is a Cy, H is incremented.

AGAIN: ADI 01H
       DAA     ;;Increment A in decimal.
       JNC SKIP ;If Cy = 0, jump to SKIP.

       INR H
       CMC     ;;If Cy = 1, increment H and reset Cy.

SKIP: DCR C     ;Decrement C.
       JNZ AGAIN;If not zero jump to AGAIN.

       ;When we come out of this loop, H and A will have the
       ;MS and LS byte respectively of the BCD result.

       MOV L, A
       SHLD CURAD
       CALL UPDAD;;;Display equivalent BCD in address field.
       CALL UPDDT  ;Display Hexadecimal number in data field.
       HLT
```

Flowchart for the second method of solving the problem is shown in Fig. 16.8b.

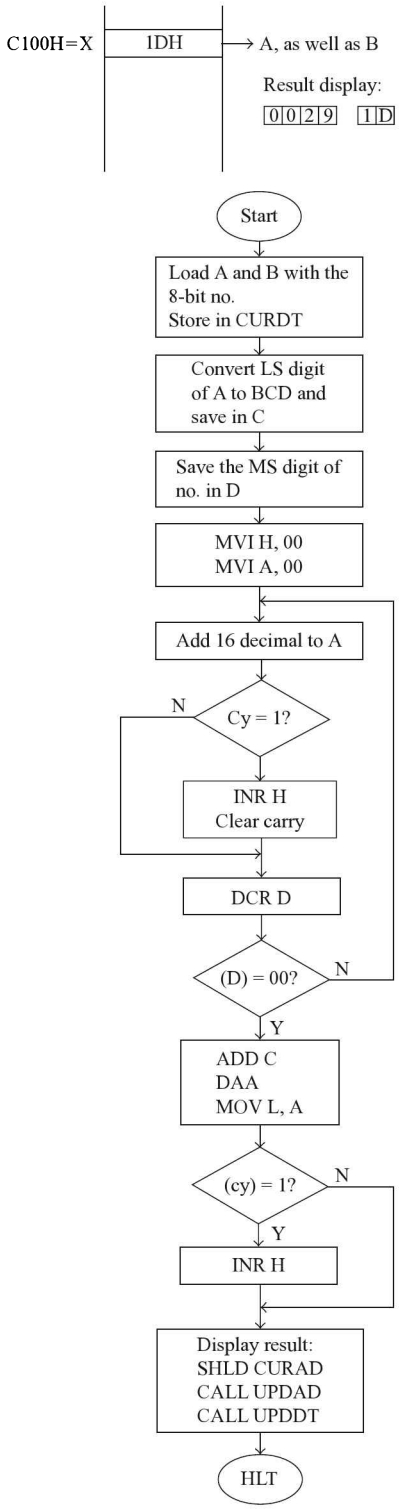


Fig. 16.8b
Flowchart to convert an 8-bit binary to BCD — Method 2

16.8.2 Program to Convert an 8-bit binary to BCD—Method 2

```
;FILE NAME C:\ALS\BINBCD2.ASM

  ;8085 ALP TO CONVERT A 8 BIT BINARY NUMBER TO
  ;EQUIVALENT BCD NUMBER. THE BINARY(HEX) NUMBER IS AT LOCATION X.
  ;DISPLAY THE BINARY(HEX) NUMBER IN THE DATA FIELD
  ;AND ITS EQUIVALENT BCD NUMBER IN THE ADDRESS FIELD.

  ;The method used is as follows. Suppose we have CF as the 8 bit
  ;hexadecimal number. It is same as C x 16 + F. Or, in decimal it
  ;is 12 x 16 + 15 = 207. So, we first convert the LS digit to BCD.
  ;Then we multiply MS digit by 16 in decimal notation, and finally
  ;add these two BCD numbers.

      ORG C100H
X:    DB 1DH

      ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
CURAD: EQU FFF7H
UPDAD: EQU 06BCH

      LDA X
      STA CURDT
      MOV B, A;;;Load A from location X. Store it in CURDT and B.

      ;The next 4 instructions convert the LS digit of A into
         ;its equivalent BCD value

      ANI 0FH ;Make MS 4 bits of A as 0000 (mask MS digit of A).
      ADI 00H ;Add 00 so that flags are affected based on A value.
      DAA     ;Decimal adjust.
      MOV C, A;Save result in C.

      ;The next 7 instructions save the MS digit of number in D

      MOV A, B;Load A from saved value in B.
      ANI F0H ;Mask LS digit of A.
      RRC
      RRC
      RRC
      RRC ;;;;Exchange LS and MS digits of A.
            ;Thus A will have MS digit of the number.
      MOV D,A;Save MS digit of the number in D.

      MVI H, 00H ;Initialise H with 00. Finally H will have
                 ;MS byte of result of conversion.
      MVI A, 00H ;Initialise A with 00. Finally A will have
                 ;LS byte of result of conversion.

      ;The next 7 instructions perform contents of D x 16 in
      ;decimal notation. Result will be in H and A.

AGAIN: ADI 16H
      DAA      ;Add 16 decimal to A.
      JNC SKIP ;If Cy = 0, jump to SKIP.
```

```
        INR H
        CMC     ;;If Cy = 1, increment H and make Cy = 0. If Cy
                ;is not reset to 0, DAA will give wrong result.
SKIP:  DCR D ;Decrement counter D.
       JNZ AGAIN ;If not zero, jump to AGAIN.

        ;When we come out of this loop, we will have in H and A
        ;MS digit of number x 16 in decimal notation.

        ;The next 5 instructions add C contents to A in decimal
        ;notation. If Cy becomes 1, then H is incremented.

        ADD C    ;Add C to A.
        DAA      ;Decimal adjust to get LS byte of BCD result.
        MOV L, A ;Save result in L.

        JNC DISP
        INR H ;;Increment H if Cy = 1.
              ;H contains MS byte of BCD result.

        ;At this point, HL will have the equivalent BCD value.

DISP:  SHLD CURAD
        CALL UPDAD;;Display BCD value in address field.
        CALL UPDDT ;Display the Hex number in data field.
        HLT
```

16.8.3 Program to Convert an 8-bit Binary to BCD—Method 3

Flowchart for the third method of solving the problem is shown in Fig. 16.8c.

```
;FILE NAME C:\ALS\BINBCD3.ASM

        ;8085 ALP TO CONVERT A 8 BIT BINARY NUMBER TO
        ;EQUIVALENT BCD NUMBER. THE BINARY NUMBER IS AT LOCATION X.
        ;DISPLAY THE BINARY NUMBER IN THE DATA FIELD
        ;AND ITS EQUIVALENT BCD NUMBER IN THE ADDRESS FIELD.

        ;The method used for conversion is as follows. Suppose we want
        ;to convert CF to equivalent BCD. First of all, we find out how
        ;many 100's are present in the number, by repeatedly subtracting
        ;64H (=100) from the number, till the number becomes less than
        ;64H. Let us say, we store this result in register H.
        ;Next, we find out how many 10's are present in the number, by
        ;repeatedly subtracting 0AH (=10) from the number, till the number
        ;becomes less than 0AH. Let us say, we store this result in
        ;register L. Once we know the number of 100's, number of 10's
        ;and the units value, we have the equivalent BCD number.

        ORG C100H
  X:    DB FFH

        ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
CURAD: EQU FFF7H
```

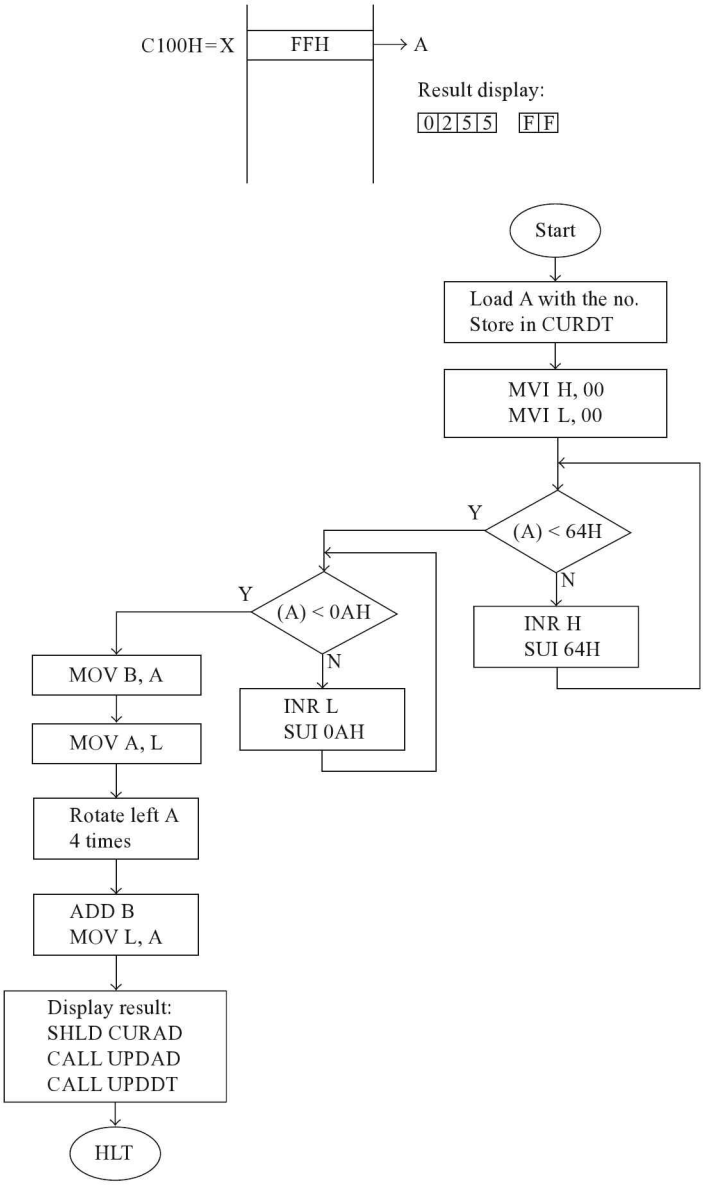


Fig. 16.8c Flowchart to convert an 8-bit binary to BCD — Method 3

```
UPDAD: EQU 06BCH

       LDA X
       STA CURDT     ;;Load A from location X. Store in CURDT.
       MVI H, 00H    ;Initialise H with 00. H will finally have the
                     ;number of 100's in the BCD result.
       MVI L, 00H    ;Initialise L with 00. L will finally have the
                     ;number of 10's in the BCD result.
```

```
        ;The next 5 instructions store in H the number of 100's.

REP1:   CPI 64H         ;Compare A with 64H i.e. 100 decimal.
        JC REP2         ;If < 100, jump to REP2 to find number of 10's.

        INR H           ;If >= 100, increment H.
        SUI 64H         ;Decrement A by 100 decimal.
        JMP REP1        ;Jump to REP1.

        ;When we are out of this loop, H will have number of 100's
        ;and A contents would have become less than 100.

        ;The next 5 instructions store in L the number of 10's.

REP2:   CPI 0AH         ;Compare A with 0AH i.e. 10 decimal.
        JC EXIT         ;If <10, jump to EXIT.

        INR L           ;If >= 10, increment L.
        SUI 0AH         ;Decrement A by 10 decimal.
        JMP REP2        ;Jump to REP2.

        ;When we are out of this loop, L will have number of 10's
        ;and A contents would have become less than 10.

EXIT:   MOV B, A        ;Save in B the units value of BCD result.

        MOV A, L        ;Load A with number of 10's in BCD result.
        RLC
        RLC
        RLC
        RLC          ;;;;Move this information to MS digit of A,
                        ;with LS digit of A as 0.

        ADD B           ;This results in A having number of 10's in MS
                        ;digit, and units value of result in LS digit.

        MOV L, A        ;Save A value in L.
                        ;Thus the total BCD result is now in HL.

        SHLD CURAD
        CALL UPDAD;;Display BCD equivalent in address field.
        CALL UPDDT ;Display Hexadecimal value in data field.
        HLT
```

■ 16.9 CHECK FOR PALINDROME

Write an 8085 assembly language program to check if the 8-bit number at location X is a palindrome or not. Display the number and the result in the address field. If it is a palindrome the result should be FF, else 00.

Flowchart for the method of solving the problem is shown in Fig. 16.9.

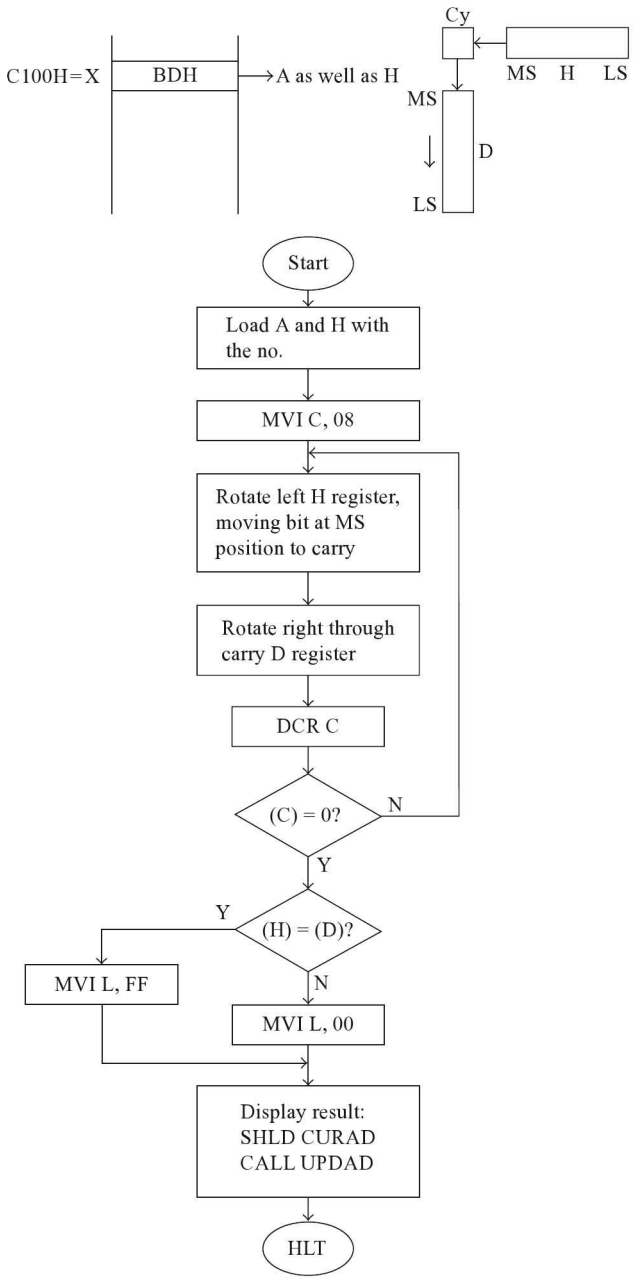


Fig. 16.9
Flowchart to check for palindrome

16.9.1 Program to Check for Palindrome

```
;FILE NAME C:\ALS\PALIN.ASM

       ;8085 ALP TO CHECK IF THE 8 BIT NUMBER AT LOCATION X
       ;IS A PALINDROME OR NOT. DISPLAY THE NUMBER AND THE RESULT IN THE
       ;ADDRESS FIELD. IF IT IS A PALINDROME THE RESULT IS FF, ELSE 00.

       ORG C100H
X:     DB BDH

       ORG C000H
CURAD: EQU FFF7H
UPDAD: EQU 06BCH

       LDA X
       MOV H, A   ;;Load H from location X.
       MVI C, 08H ;Load C with 8. C is used as down counter.

AGAIN: MOV A,H
       RLC
       MOV H, A ;;;Rotate left contents H. A bit of H register
                  ;moved to Cy starting with MS bit.

       MOV A, D
       RAR
       MOV D, A ;;;Rotate right through Carry contents of D. The bit moved
                  ;out of H register enters D register through the MS bit.

       DCR C      ;Decrement C.
       JNZ AGAIN  ;If not zero, jump to AGAIN.

       ;By the time we come out of the loop, D register will have
       ;the reverse of the byte in location X. And because of 8
       ;rotations, H register will have the original number.

       MOV A, H
       CMP D   ;;Compare the number and its reverse.
       JZ PALIN ;If they are same, jump to PALIN.

       MVI L, 00H ;Load L with 00, if the byte is not a palindrome.
       JMP EXIT   ;Jump to EXIT to display result.

PALIN: MVI L,FFH ;Load L with FF, if the byte is not a palindrome.

EXIT:  SHLD CURAD
       CALL UPDAD;;Display result in the address field.
       HLT
```

■ 16.10 COMPUTE THE LCM OF TWO 8-BIT NUMBERS

Write an 8085 assembly language program to compute the LCM of two 8-bit numbers stored in locations X and Y. Display the result in the address field.

Flowchart for the method of solving the problem is shown in Fig. 16.10.

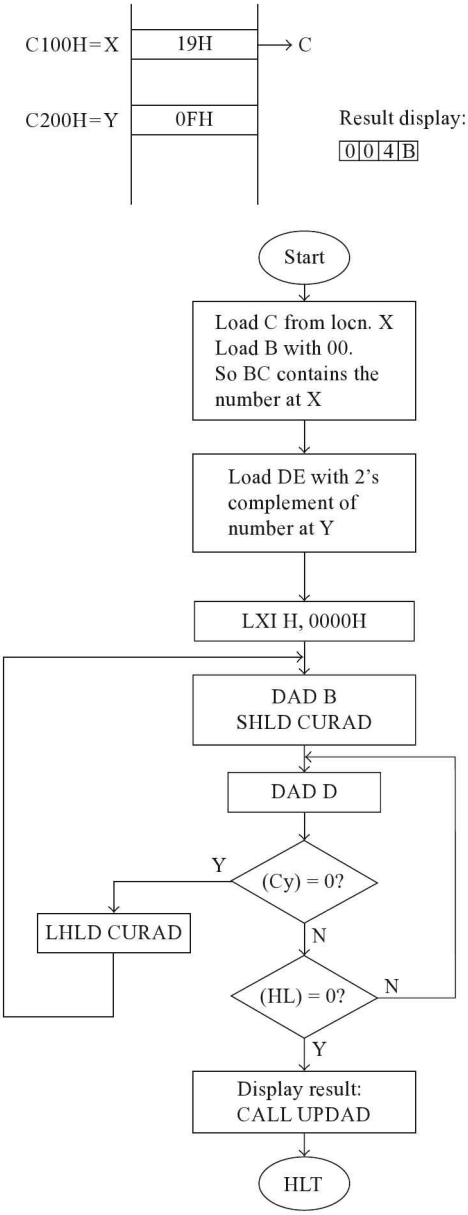


Fig. 16.10
Flowchart to compute the LCM of two bytes

16.10.1 Program to Compute LCM

```
;FILE NAME C:\ALS\LCM.ASM

        ;8085 ALP TO DETERMINE THE LCM OF TWO 8 BIT NUMBERS STORED IN
        ;LOCATIONS X AND Y. DISPLAY THE RESULT IN THE ADDRESS FIELD

        ;The method used is as follows. Let us say, the two numbers are
        ;25 and 15. We divide the first number by the second number. If
        ;there is no remainder, then the first number is the LCM. But in
```

```
        ;this case, there is a remainder. So we try to see if the next
        ;multiple of 25, i.e. 2 x 25 = 50 is the LCM. Again, 50 does not
        ;divide exactly by 15, and so 50 is not the LCM. Next we try if
        ;3 x 25 = 75 is the LCM. 75 divides exactly by 15. Thus 75 is
        ;the LCM of 25 and 15.

        ORG C100H
   X:   DB 19H

        ORG C200H
   Y:   DB 0FH

        ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH

        LXI H, X
        MOV C, M
        MVI B, 00;;;Load BC with the 8 bit number at location X.

        ;The next 5 instructions load DE with the 2's complement
        ;of number at Y, i.e. negative of the number at Y.

        LDA Y
        CMA
        MOV E, A ;;;Load E with 1's complement of the number at Y.
        MVI D, FFH ;Now DE contains 1's complement of number at Y.
        INX D      ;DE contains 2's complement of the number at Y.

        LXI H, 0   ;Initialise HL with 0. Finally HL will have LCM.

NEXT:  DAD B       ;Add to HL the first number. Every time through
                   ;the loop, HL will be loaded with the next
                   ;multiple of the first number.

       SHLD CURAD;Store result in word location CURAD.

        ;The next 6 instructions perform the following. It repeatedly
        ;subtracts the second number from HL till result becomes zero or
        ;negative. If result is zero, jumps to EXIT, and if negative
        ;jumps to SKIP. Note that result is negative, if Cy = 0.

AGAIN: DAD D    ;Subtract from HL the second number.
        JNC SKIP ;If result is negative, jump to SKIP.

        MOV A, H
        ORA L
        JZ EXIT ;;;If result in HL is 0000H, jump to EXIT.

        ;DAD instruction does not affect flags except Cy. So
        ;we have used indirect method to check for zero result.

        JMP AGAIN ;If result is positive, jump to AGAIN.

        ;The next 2 instructions result in reloading HL from CURAD
        ;and jumping to NEXT, so that HL will have the next multiple
        ;of the first number.
```

```
SKIP:  LHLD CURAD
       JMP NEXT ;;Reload HL from CURAD and jump to NEXT.

       ;At this point we have in HL and word location CURAD, the LCM.

EXIT:  CALL UPDAD ;Display LCM in the address field.
       HLT
```


■ 16.11 SORT NUMBERS USING BUBBLE SORT

Write an 8085 assembly language program to sort N 1-byte binary numbers in ascending order, using bubble sort. N is stored at location X, and the numbers from location X+1.
Flowchart for the method of solving the problem is shown in Fig. 16.11.

16.11.1 Program to Perform Sorting Using Bubble Sort

```
;FILE NAME C:\ALS\BUBSRT.ASM

       ;8085 ALP TO SORT N ONE BYTE BINARY NUMBERS IN ASCENDING
       ;ORDER USING BUBBLE SORT. N IS STORED AT LOCATION X, AND
       ;THE NUMBERS FROM LOCATION X+1.

       ;MODIFY THE SAME PROGRAM TO SORT IN DESCENDING ORDER.

       ORG C100H
  X:   DB 04H,33H,22H,44H,11H

       ORG C000H

       LXI H, X
       MOV C, M     ;;Load C from memory location X.
       DCR C         ;Decrement C. C now has the number
                     ;of passes still to be performed.

OUTLOOP:
       MVI E, 01H    ;Load E with 1. E contains one more than the
                     ;number of exchanges made in a pass so far.

       MOV B, C      ;Load B from C contents. B now has the number
                     ;of comparisons still to be performed in a pass.

       INX H         ;Point HL to the first element of the array.

INLOOP:
       MOV A, M      ;Load A from memory pointed by HL.
       INX H         ;Point HL to next element of array.
       CMP M         ;Compare A (previous element) and
                     ;next element of the array.

       JC SKIP       ;If (previous element is < next element), then
                     ;jump to SKIP, without performing exchange.

       ;CHANGE 'JC SKIP' TO 'JNC SKIP' FOR SORTING IN DESCENDING ORDER

  ;The following 5 instructions perform exchange of previous and
```

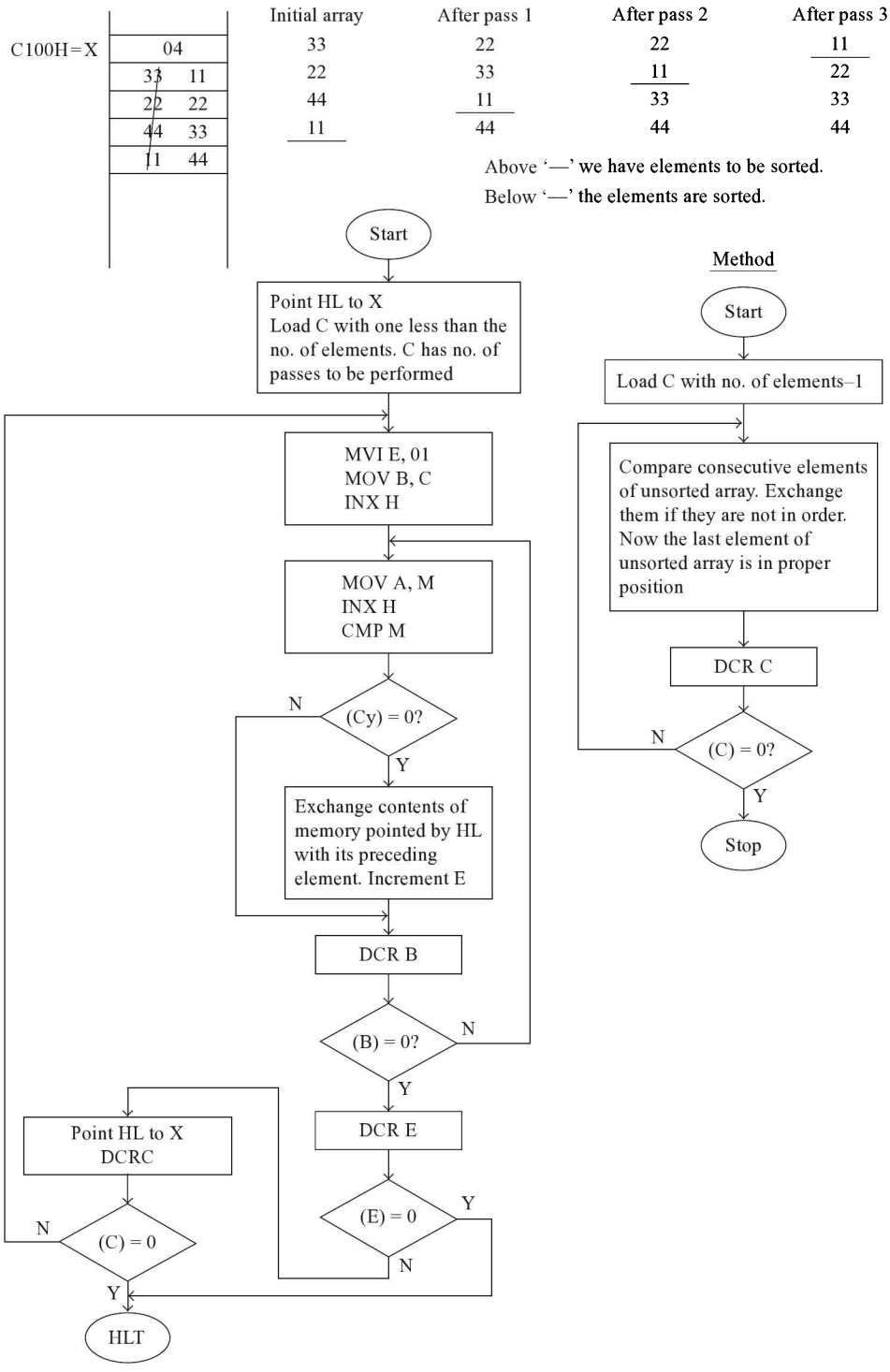


Fig. 16.11 Flowchart to perform sorting using bubble sort

```
;next elements.

      MOV D, M      ;Save next element in D register.
      MOV M, A      ;Load in next element position the previous element.
      DCX H         ;Point HL to previous element.
      MOV M, D      ;Previous element position loaded with value in D.
      INX H         ;Point HL to next element.

      INR E         ;Increment E, as an exchange has taken place.

SKIP: DCR B         ;Decrement B, as a comparison in a pass is over.
      JNZ INLOOP    ;If non zero, jump to INLOOP to
                    ;perform next comparison.

      ;When we are out of this inner loop, it means we have
      ;completed a pass of the array. Then we check if any
      ;exchanges were made during the pass.

      DCR E         ;Decrement E
      JZ EXIT       ;If E value is zero, it means no exchanges were
                    ;made in this pass. So jump to EXIT.

      LXI H, X      ;Load HL with X. From X+1, the array begins.
      DCR C         ;Decrement C, as a pass through the array is over.
      JNZ OUTLOOP   ;If non zero, jump to OUTLOOP to
                    ;perform next pass.

EXIT: HLT           ;Halt, as the sorting is complete.
```

■ 16.12 SORT NUMBERS USING SELECTION SORT

Write an 8085 assembly language program to sort N 1-byte binary numbers in ascending order, using selection sort. N is stored at location X, and the numbers from location X+1.

Flowchart for the method of solving the problem is shown in Fig. 16.12.

16.12.1 Program to Perform Sorting Using Selection Sort

```
;FILE NAME C:\ALS\SELSORT.ASM

;8085 ALP TO SORT N ONE BYTE BINARY NUMBERS IN ASCENDING
;ORDER USING SELECTION SORT. N IS STORED AT LOCATION X, AND
    ;THE NUMBERS FROM LOCATION X+1.

    ;MODIFY THE SAME PROGRAM TO SORT IN DESCENDING ORDER

    ORG C000H
X:  EQU C100H

    ;In this program, we have not indicated data values. The user is
    ;required to provide them, using the kit in keyboard mode, or
    ;using the PC in serial mode.

        ;NOTE: It is assumed for simplicity that
        ;X is an address with last 2 digits as 00H.
```

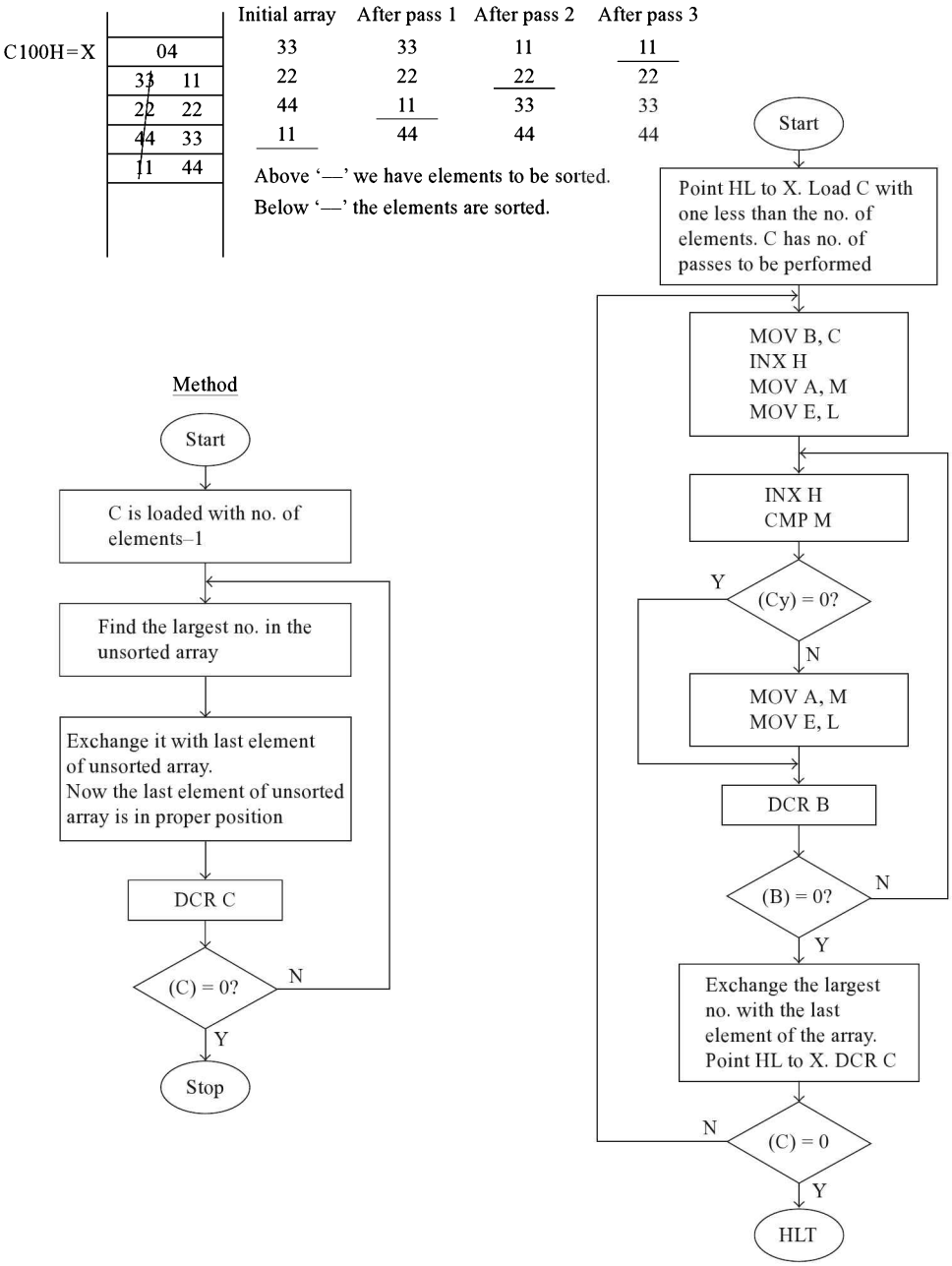


Fig. 16.12 Flowchart to perform sorting using selection sort

```
        LXI H, X
        MOV C, M     ;;Load C from location X.
        DCR C         ;Decrement C. C indicates the number of
                      ;passes still to be made.

OUTLOOP:
        MOV B, C    ;Load B from C. B indicates the number of
                    ;comparisons still to be made in a pass.
        INX H       ;Point HL to the first element of the array.
        MOV A, M    ;Load A with the first element of the array. By the
                    ;end of the pass, A will have the largest element.
        MOV E, L    ;Load E from L. E indicates the position in
                    ;the array of the element in A.

INLOOP:INX H        ;Point HL to next element of array.
        CMP M       ;Compare A (previous element) and next element.
        JNC SKIP    ;If (A >= next element) jump to SKIP.

        ;CHANGE 'JNC SKIP' TO 'JC SKIP' FOR SORTING IN DESCENDING ORDER

        MOV A, M
        MOV E, L    ;;If (A < next element), load A with next element,
                    ;and E with position of next element.

SKIP:   DCR B       ;Decrement B, as a comparison is over.
        JNZ INLOOP  ;If nonzero, perform one more comparison.

        ;At this point, a pass is completed. A has the largest element,
        ;and E has the position in the array of the largest element. The
        ;next 4 instructions exchange the largest element in the array
        ;with the last element of the array.

        MOV D, M ;Save in D, the last element of the array.
        MOV M, A ;Last location of array loaded with A.
        MOV L, E ;Load HL with saved position of the largest element.
        MOV M, D ;Load this memory with the saved value in D.

        LXI H, X      ;Load HL with X. From X+1, the array begins.
        DCR C         ;Decrement C, as a pass is over.
        JNZ OUTLOOP   ;If nonzero, perform one more pass.
        HLT           ;Halt, as sorting is over.
```

■ 16.13 SIMULATE DECIMAL UP COUNTER

Write an 8085 assembly language program to perform as a decimal up counter (from 00 to 99). The count should be incremented every half second, and is to be displayed in the data field.

Flowchart for the method of solving the problem is shown in Fig. 16.13.

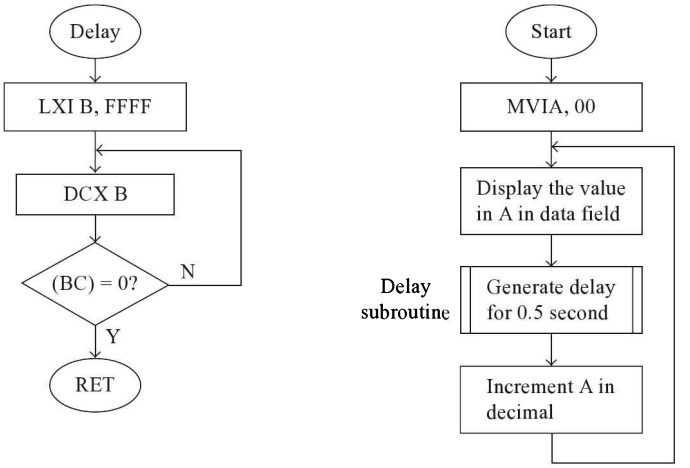


Fig. 16.13 Flowchart to simulate decimal up counter

16.13.1 Program to Simulate Decimal Up Counter

```
;FILE NAME C:\ALS\UPCNTR.ASM

       ;8085 ALP TO PERFORM AS A DECIMAL UP COUNTER (FROM 00 TO 99).
       ;THE COUNT SHOULD BE INCREMENTED EVERY HALF SECOND, AND
       ;THE COUNT IS TO BE DISPLAYED IN THE DATA FIELD, AND THE
       ;OPERATION SHOULD REPEAT ENDLESSLY.

       ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
       MVI A, 00H    ;Initialise A with 00.

REP:   STA CURDT     ;Store A value in CURDT.
       CALL UPDDT    ;Display in the data field contents of CURDT.

       CALL DELAY    ;Generate a delay of 0.5 seconds

       LDA CURDT     ;Reload A from location CURDT.
       ADI 01H
       DAA           ;;Increment A in decimal notation.
       JMP REP       ;Jump to REP.
  ;------ Subroutine to generate a delay of 0.5 second. Its
  ;------ working is explained at the end of this program.

DELAY: LXI B, FFFFH ;Load BC with FFFFH.
AGAIN: DCX B        ;Decrement BC. It does not affect flags.
       MOV A, B
       ORA C        ;So, perform OR of B and C registers.
       JNZ AGAIN    ;If not zero, jump to AGAIN.
       RET          ;Return, after generating delay of 0.5 second.
```

Note: Replace 'JMP REP' with the following three instructions, if it is required to stop after counting upto 99.

```
       CPI 00H
       JNZ REP
       HLT
```

16.13.2 Generation of Time Delay

There are times when we are required to generate time delays. In this example, we want the count value to change once every half second. Later in the chapter, we simulate a real time clock, where the time display should change once every second. Such time delays can be generated by a microprocessor. The principle behind time delay generation by the microprocessor is as follows. Execution of every instruction needs some clock cycles. Thus, to generate the required time delay, a microprocessor will have to execute the required number of instructions. This is the principle of time delay generation by the microprocessor.

NOP instructions for time delay: The immediate reaction will be to use NOP instructions to generate the required time delay. Each NOP instruction uses four clocks for fetching, decoding, and executing. In the discussion that follows, it is assumed that 8085 is working with a crystal frequency of 6 MHz, and as such with an internal frequency of 3 MHz. Thus each clock period is one-third of a microsecond. In such a case, a NOP instruction needs only 1.33 μs for execution. Even if we use 64K NOP instructions, which is the maximum possible in memory, the time delay would only be 64K×1.33 μs = about 86 μs. Also, the program size has become too much, for too little work done. Thus, we use a series of NOP instructions only when our interest is to generate small time delays of a few microseconds.

Use of an 8-bit register as counter in a loop: A much better way to generate a larger time delay with a smaller program size, is to execute a series of instructions repetitively in a loop. An example is shown below.

```
      MVI C, FFH   ;Uses 7 T states.
REP:  DCR C        ;Uses 4 T states.
      JNZ REP      ;Uses 10 T states in the case of jump
                   ;and 7 T states in the case of no jump.
      RET          ;Uses 10 T states.
```

In this example, C is loaded with maximum possible value of FFH. Then, C is decremented. If the result is non-zero, the decrement operation is repeated. The program is quite short, and generates a delay of 1.191 μs, as explained below.

By the time we come out of the loop, the number of T states used up will be

$$\begin{aligned} &7 + [(4 + 10) * FF] - 3 + 10 \\ &= 14 + 14 * FF \\ &= 3584 \end{aligned}$$

where −3 is used to take into account only seven *T* states used instead of ten *T* states for the JNZ REP instruction when C is decremented to 00. Thus the time delay generated is 3584 * 1/3 μs = 1195 μs.

If we need delays smaller than 1195 μs, we load the count register with a proportionately small value. If we need larger delay than 1195 μs, we have to use a loop inside a loop as shown in the following.

```
      MVI B, FFH
REP2: MVI C, FFH
REP1: DCR C
      JNZ REP1
      DCR B
      JNZ REP2
      RET
```

The user is left to calculate the delay generated in this case. It will be about 305 μs. The logic can be extended further to generate any amount of delay.

Use of a register pair as counter in a loop: Instead of using an 8-bit register, we can use a register pair as counter in a loop, to generate much larger time delays using smaller program size. This is the method used in the above program to generate a half-second delay. The calculation for the delay is shown in the following.

```
DELAY:LXI B,FFFFH  ;Uses 10 T states
AGAIN:DCX B        ;Uses 6 T states
      MOV A, B     ;Uses 4 T states
      ORA C        ;Uses 4 T states
      JNZ AGAIN    ;Uses 10 T states for jump case
                   ;and 7 T states for no jump case
      RET          ;Uses 10 T states
```

Number of T states, $N = 10 + (6 + 4 + 4 + 10) * \text{FFFFH} - 3 + 10$
$= 17 + 24 * \text{FFFFH} = 1572857$

Thus, time delay = 1572857 * 1/3 μs = 0.52428 s.

This delay is generally taken as about a half-second delay. The user is left with the exercise to compute the count value to be loaded in the register pair for exact 0.5-s delay.

However, if we use the microprocessor to generate large time delays, the microprocessor will not be available for any other operation for the entire delay time. So it is advisable to restrict the time delay generation by the microprocessor to only a few micro- or milliseconds in practical applications. For large time delays, it is better to use dedicated timer chips like Intel 8253, which is discussed in a later chapter.

■ 16.14 SIMULATE DECIMAL DOWN COUNTER

Write an 8085 assembly language program to perform as a decimal down counter (from 99 to 00). The count should be decremented every half second, and the count is to be displayed in the data field.

Flowchart for the method of solving the problem is shown in Fig. 16.14.

16.14.1 Program to Simulate Decimal Down Counter

```
;FILE NAME C:\ALS\DNCNTR.ASM

       ;8085 ALP TO PERFORM AS A DECIMAL DOWN COUNTER (FROM 99 TO 00).
       ;THE COUNT SHOULD BE DECREMENTED EVERY HALF SECOND, AND
       ;THE COUNT IS TO BE DISPLAYED IN THE DATA FIELD, AND THE

       ;OPERATION SHOULD REPEAT ENDLESSLY.

       ORG C000H
CURDT: EQU FFF9H
UPDDT: EQU 06D3H
```

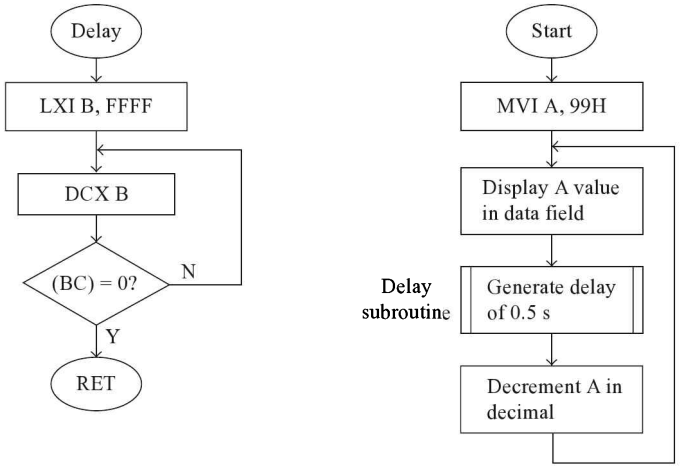


Fig. 16.14 Flowchart to simulate decimal down counter

```
        MVI A, 99H    ;Initialise A with 99.
REP:    STA CURDT     ;Store A value in CURDT.
        CALL UPDDT    ;Display contents of CURDT in data field.
        CALL DELAY    ;Generate delay of 0.5 second.
        LDA CURDT     ;Reload A from location CURDT.
        ADI 99H
        DAA           ;;Decrement A in decimal notation by
                      ;adding 99, which is 10's complement of 01.
        JMP REP       ;Jump to REP to display the next count.
        ;------ Subroutine to generate a delay of 0.5 second.
DELAY:  LXI B,FFFFH
AGAIN:  DCX B
        MOV A, B
        ORA C
        JNZ AGAIN
        RET
```

Note: Replace 'JMP REP' with the following three instructions, if it is required to stop after counting down to 00.

```
        CPI 99H
        JNZ REP
        HLT
```

■ 16.15 DISPLAY ALTERNATELY 00 AND FF IN THE DATA FIELD

Write an 8085 assembly language program to alternately display 00 and FF in the data field, with a delay of N seconds, where N is less than or equal to 2.

Flowchart for the method of solving the problem is shown in Fig. 16.15.

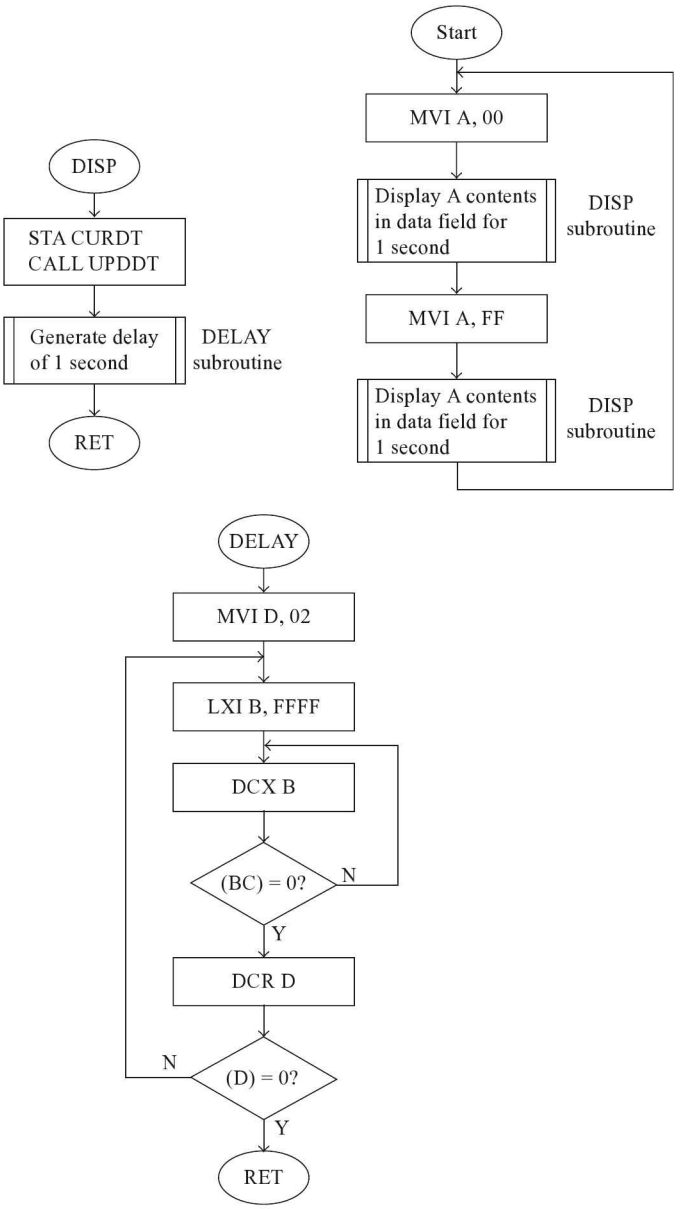


Fig. 16.15 To alternately display 00 and FF in the data field

16.15.1 Program to Alternately Display 00 and FF in the Data Field

```
;FILE NAME C:\ALS\DISP00FF.ASM

       ;8085 ALP TO DISPLAY 00 AND FF ALTERNATELY WITH
       ;A DELAY OF N SECONDS, WHERE N <=2.

       ORG C000H
```

```
CURDT: EQU FFF9H
UPDDT: EQU 06D3H

REPT:  MVI A,00H
       CALL DISP     ;;Display 00 in the data field for 1 second.
       MVI A,FFH
       CALL DISP     ;;Display FF in the data field for 1 second.
       JMP REPT      ;Jump to REPT, to repeat the operations.

       ;The following subroutine displays the contents of Accumulator,
       ;in the data field for about 1 second.

DISP:  STA CURDT
       CALL UPDDT
       CALL DELAY
       RET

       ;The following subroutine generates a delay of about half the
       ;value present in D register. Thus, it generates 1-second delay.

DELAY: MVI D, 02H

LOOP1: LXI B, FFFFH
LOOP2: DCX B
       MOV A, B
       ORA C
       JNZ LOOP2  ;;;;;Generate delay of 0.5 second.

       DCR D          ;Decrement D.
       JNZ LOOP1      ;If nonzero, jump to LOOP1.
       RET
```

■ 16.16 SIMULATE A REAL-TIME CLOCK

Write an 8085 assembly language program to simulate a real time clock, which displays hours and minutes in the address field, and seconds in the data field, using 24-hour format.

Flowchart for the method of solving the problem is shown in Fig. 16.16.

16.16.1 Program to Simulate a Real-time Clock

```
;FILE NAME C:\ALS\REAL_CLK.ASM

       ;8085 ALP TO GENERATE REAL TIME CLOCK, WHICH DISPLAYS
       ;HOURS AND MINUTES IN THE ADDRESS FIELD AND SECONDS IN THE
       ;DATA FIELD (USING 24 HOUR FORMAT).

       ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
CURAD: EQU FFF7H
UPDAD: EQU 06BCH

BEGIN: LXI H,0000H ;Initialise HL with 0000. H and L indicate
```

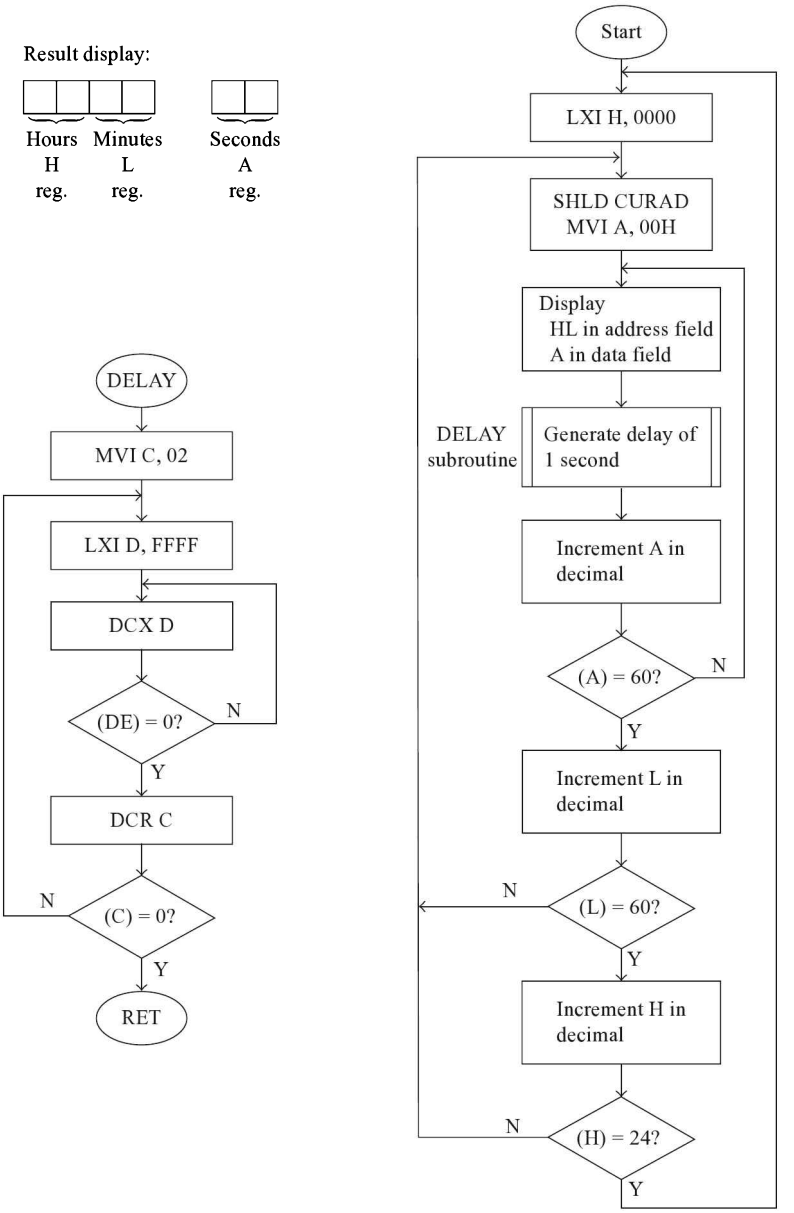


Fig. 16.16 Flowchart to simulate a real-time clock

```
                    ;Hours and minutes respectively.
HR_MIN:
       SHLD CURAD ;Store HL in word location CURAD.
       MVI A, 00H ;Initialise A with 00. A indicates seconds.
NXT_SEC:
       STA CURDT  ;Store A value in location CURDT.
       CALL UPDAD ;Display Hours and Minutes in address field.
```

```
        CALL UPDDT ;Display Seconds in the data field.
        CALL DELAY ;Generate delay of 1 second.

        LDA CURDT  ;Reload A from CURDT.
        ADI 01H
        DAA          ;;Increment A in decimal notation.
        CPI 60H       ;Compare A and 60.
        JNZ NXT_SEC   ;If A is not equal to 60, jump to NXT_SEC.

        ;When we come out of this loop, A will be 60. In such a case,
        ;it is necessary to make A as 00, and increment the Minute value,
        ;which is present in L.

        LHLD CURAD ;Reload HL from CURAD.

        MOV A, L
        ADI 01H
        DAA
        MOV L, A ;;;;Increment L in decimal notation.
        CPI 60H
        JNZ HR_MIN ;;If L value is not equal to 60, jump to HR_MIN.
                    ;A will be made 00, after jumping to HR_MIN.

        ;When we are at this point, L will be 60. In such a case, it is
        ;necessary to make L as 00, and increment the Hours value in H.

        MVI L, 00H ;Make L as 00.
        MOV A, H
        ADI 01H
        DAA
        MOV H, A     ;;;;Increment H in decimal.

        CPI 24H
        JNZ HR_MIN    ;;If H value is not equal to 24, jump to HR_MIN.
        JMP BEGIN      ;If H value is 24, jump to BEGIN.

        ;------ Subroutine DELAY to generate a time delay of 1 second

DELAY: MVI C,02H
OUTLOOP:
        LXI D, FFFFH

INLOOP:
        DCX D
        MOV A, D
        ORA E
        JNZ INLOOP

        DCR C
        JNZ OUTLOOP
        RET
```

Note: To test the program fully with a 1-s delay, we need 24 hours! To solve this problem, we can test the working of the seconds display, with seconds display getting updated every second. Then we change the delay such that the minutes field gets updated every second. Thus, the minutes display can be tested in just 60 more seconds. Then we change the delay such that the hours field gets updated every second. Thus, the hours display can be tested in just 24 more seconds.

To change the display in the minutes field approximately once every second, load DE pair with 0400H in this subroutine. To change the display in the hours field approximately once every second, load DE pair with 0010H in this subroutine.

1. Write an 8085 assembly language program to find the largest and smallest numbers in an array of unsigned numbers. Display them in the address field.
2. Write an 8085 assembly language program to convert a two-digit octal number to BCD. Display the result in the address field.
3. Write an 8085 assembly language program to convert a two-digit BCD number to octal. Display the BCD number in the data field, and the octal number in the address field.
4. Write an 8085 assembly language program to work as up counter to count in octal from 00 to 77 octal, the display in the data field changing once every second.
5. Write an 8085 assembly language program to generate a delay of:

   a. Exactly 0.5 second;
   b. Exactly 1.0 second;

   assuming that the internal frequency of operation is 3 MHz.
6. Write an 8085 assembly language program to convert ASCII hex value to binary, using look up table approach. Display ASCII value and equivalent binary in the address field.
7. Write an 8085 assembly language program to convert a two-digit hex value to two equivalent ASCII codes, using look up table approach. Display the hex value in data field and equivalent ASCII value in the address field.
8. Write an 8085 assembly language program to find the LCM of two given 8-bit numbers, using a method other than that described in this chapter. Display the result in the address field.
9. Write an 8085 assembly language program to check if a given byte is a palindrome or not, using a method other than that described in this text. Display the result in the address field.
10. Write an 8085 assembly language program to simulate a real-time clock, which displays hours, minutes, and seconds in 12-hour format. It should display a dot at the end of the data field if the time is afternoon.

17

More Complex Assembly Language Programs

■ Subtract multi-byte BCD numbers

● *Program for subtraction of multi-byte BCD numbers*

■ Convert 16-bit binary to BCD

● *Program to convert a 16-bit binary number to BCD*

■ Do an operation on two numbers based on the value of X

● *Program to do an operation on two numbers based on the contents of X*

■ Do an operation on two BCD numbers based on the value of X

● *Program to do an operation on two BCD numbers based on the contents of X*

■ Bubble sort in ascending/descending order as per choice

● *Program to perform bubble sort in ascending/descending order*

● *Alternative program to perform bubble sort based on choice*

■ Selection sort in ascending/descending order as per choice

● *Program to perform selection sort in ascending/descending order*

■ Add contents of *N* word locations

● *Program to add the contents of N word locations*

■ Multiply two 8-bit numbers (shift and add method)

● *Program to multiply two 8-bit numbers (shift and add method)*

■ Multiply two 2-digit BCD numbers

● *Program to multiply two 2-digit BCD numbers*

■ Multiply two 16-bit binary numbers

● *Program to multiply two 16-bit binary numbers*

■ Questions

So far simple assembly language programs have been discussed. In this chapter, comparatively more complex programs and their flowcharts are dealt with. These include programs to subtract multi-byte BCD numbers, to convert 16-bit binary to BCD, bubble sort and selection sort in ascending/descending order, to multiply two 8-bit numbers, to multiply 2-digit BCD numbers, etc.

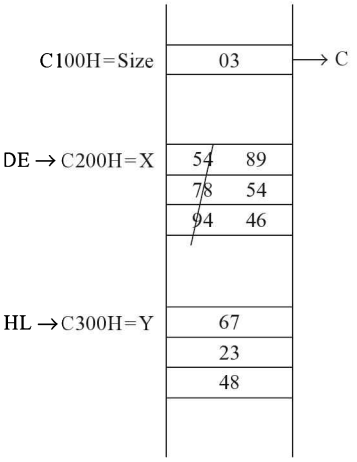
■ 17.1 SUBTRACT MULTI-BYTE BCD NUMBERS

Write an 8085 assembly language program to subtract two multi-byte BCD numbers. The numbers are stored from locations X and Y in byte reversal form. The size (in bytes) of the multi-byte numbers is provided at location SIZE. The number starting at Y is subtracted from that starting at X. The result is stored in locations starting from X.

Flowchart for solving the problem is shown in Fig. 17.1.

17.1.1 Program for Subtraction of Multi-byte BCD Numbers

```
  ;FILE NAME C:\ALS\SBLRGBCD.ASM

  ;8085 ALP TO PERFORM MULTI BYTE BCD SUBTRACTION. THE SIZE
  ;IN BYTES, OF THESE MULTI BYTE NUMBERS IS STORED AT LOCATION SIZE.
  ;THE NUMBERS ARE STORED IN LOCATIONS STARTING FROM X AND Y RESPECTIVELY.
  ;THE NUMBER STARTING AT Y IS SUBTRACTED FROM THE NUMBER
  ;STARTING AT X. RESULT IS STORED IN LOCATIONS STARTING FROM X.

  ;IF THE NUMBER STARTING AT LOCATION Y IS THE LARGER ONE, THE
  ;RESULT WILL BE IN 10'S COMPLEMENT FORM.

       ORG C100H
SIZE:  DB 03H

       ORG C200H
   X:  DB 56H, 78H, 94H

       ORG C300H
   Y:  DB 67H, 23H, 48H

       ORG C000H

       LXI H, SIZE
       MOV C, M ; (C) = 3. C register is used as loop counter.
       LXI D, X ; (DE) = C200. DE points to Minuend.
       LXI H, Y ; (HL) = C300. HL points to subtrahend.

  ;The execution of the following 4 instructions load A with 10's
  ;complement of the LS byte of subtrahend the first time through the
  ;loop. In other passes through the loop, it loads A with 9's
  ;complement of the higher bytes of subtrahend plus carry generated by
  ;previous addition.

       STC          ; (Cy) = 1
AGAIN: MVI A, 99H   ; (A)  = 99 | 99   | 99
       ACI 00H      ; (A)  = 9A | 99   | 9A
       SUB M        ; (A)  = 33 | 76   | 52
```



	94	78	56			94	78	56
−	48	23	67	Same as the addition →	+	51	76	33
	46	54	89			46	54	89

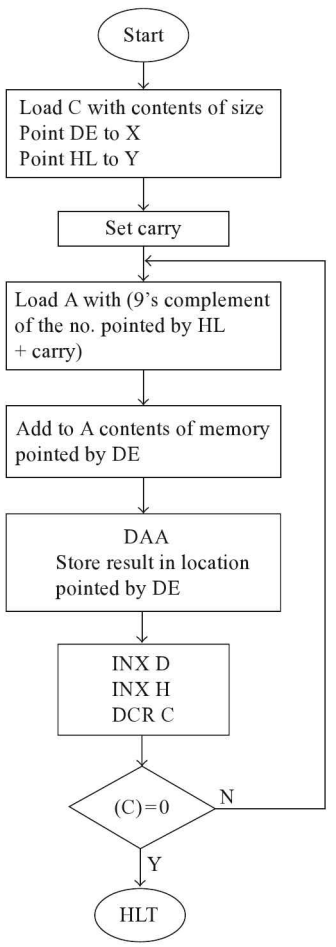


Fig. 17.1 Flowchart for the subtraction of multi-byte BCD numbers

```
;The next 4 instructions adds the Accumulator contents to
;minuend byte, performs decimal adjustment, and stores result.

     XCHG      ;(HL) = C200    |C201   |C202
     ADD M     ;   (A) = 89    | EE    | E6
     DAA       ;   (A) = 89    | 54    | 46
               ;   (Cy) = 0    | 1     | 1
     MOV M, A  ;(C200) = 89    |(C201)=54|(C202)=4

     ;Increment the address pointers after exchange of HL with DE

     XCHG
     INX H ;(HL) = C301        | C302 | C303
     INX D ;(DE) = C201        | C202 | C203

     ;Decrement C register, and branch to AGAIN if (C)<>00.

     DCR C     ; (C) = 02      | 01   | 00
     JNZ AGAIN ;         J     |  J   | NJ
     HLT
```

■ 17.2 CONVERT 16-BIT BINARY TO BCD

Write an 8085 assembly language program to convert a 16-bit binary number to BCD. Display the binary number in the address field for a second, and then display the BCD value using all the six digits of display.

Flowchart for solving the problem is shown in Fig. 17.2.

17.2.1 Program to Convert a 16-bit Binary Number to BCD

```
;FILE NAME C:\ALS\BIN16BCD.ASM

  ;8085 ALP TO CONVERT A 16 BIT BINARY NUMBER TO BCD. THE NUMBER IS AT
  ;LOCATIONS X AND X+1. DISPLAY THE NUMBER IN THE ADDRESS FIELD FOR
  ;A SECOND, AND THEN DISPLAY BCD EQUIVALENT IN ADDRESS AND DATA FIELD.

UPDAD: EQU 06BCH
UPDDT: EQU 06D3H
CURAD: EQU FFF7H
CURDT: EQU FFF9H
DELAY: EQU 04BEH

       ORG C100H
    X: DW FFFFH

       ORG C000H

       LHLD X
       SHLD CURAD
       CALL UPDAD ;;;Display the binary number in address field.

       LXI D, FFFFH
```

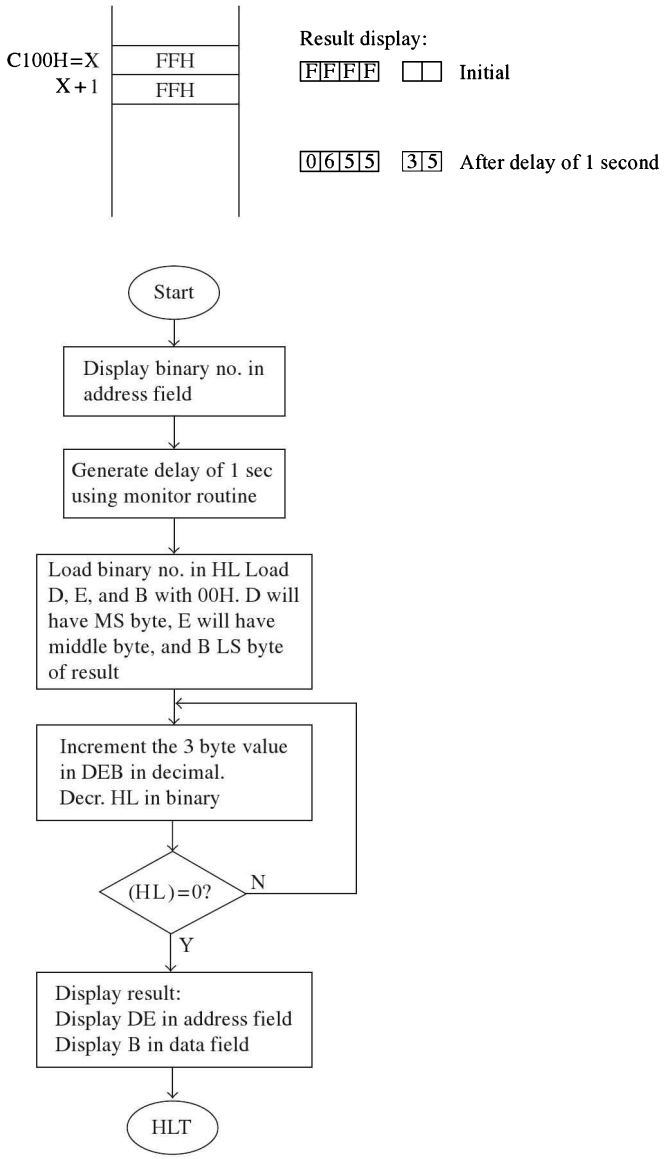


Fig. 17.2
Flowchart for converting a 16-bit binary number to BCD

```
        CALL DELAY
        CALL DELAY ;;;Generate a delay of 1 second.
        LHLD X       ;Load HL with the binary number.
        LXI D, 0000H ;Initialise DE with 0000.
        MVI A, 00H   ;Initialise A with 00.
AGAIN:  ADI 01H
        DAA          ;;Increment A in decimal.
        MOV B, A     ;Save A value in B register.
        JNC SKIP     ;If A value is <= 99, jump to SKIP.
        MOV A, E
        ADI 01H
```

```
        DAA
        MOV E, A ;;;;If A = 00 and Cy = 1, increment E in decimal.
        JNC SKIP    ;If E value is <= 99, jump to SKIP.

        MOV A, D
        ADI 01H
        DAA
        MOV D, A ;;;;If E = 00 and Cy = 1, increment D in decimal.

SKIP:   DCX H
        MOV A, H
        ORA L
        MOV A, B
        JNZ AGAIN ;;;;;Decrement HL. If nonzero jump to AGAIN
                      ;after loading A from saved value in B.

;When we come out of the loop, DE and A will have the BCD
;value, with A having the LS byte.

        XCHG
        SHLD CURAD ;;Store DE contents in word location CURAD.
        STA CURDT
        CALL UPDDT ;;Display A contents in data field.
        CALL UPDAD  ;Display DE contents in address field.
        HLT
```

■ 17.3 DO AN OPERATION ON TWO NUMBERS BASED ON THE VALUE OF X

Write an 8085 assembly language program to perform an operation on two binary numbers at X+1 and X+2, based on the contents of X. The operation to be performed is addition, subtraction, or multiplication if the contents of X are 00, 01, or 02, respectively.

Flowchart for solving the problem is shown in Fig. 17.3.

17.3.1 Program to Do an Operation on Two Numbers Based on the Contents of X

```
;FILE NAME C:\ALS\BINFUNC.ASM

  ;8085 ALP TO PERFORM THE FOLLOWING FUNCTIONS BASED ON THE CONTENTS
  ;OF LOCATION X.

    ;IF CONTENTS OF X IS 00H, PERFORM ADDITION OF NEXT TWO LOCATIONS.
    ;IF CONTENTS OF X IS 01H, PERFORM SUBTRACTION.
    ;IF CONTENTS OF X IS 02H, PERFORM MULTIPLICATION.
    ;DISPLAY THE RESULT IN THE ADDRESS FIELD IN EACH CASE.

      ORG C100H
  X:  DB 00H, 97H, 88H

      ORG C000H
```

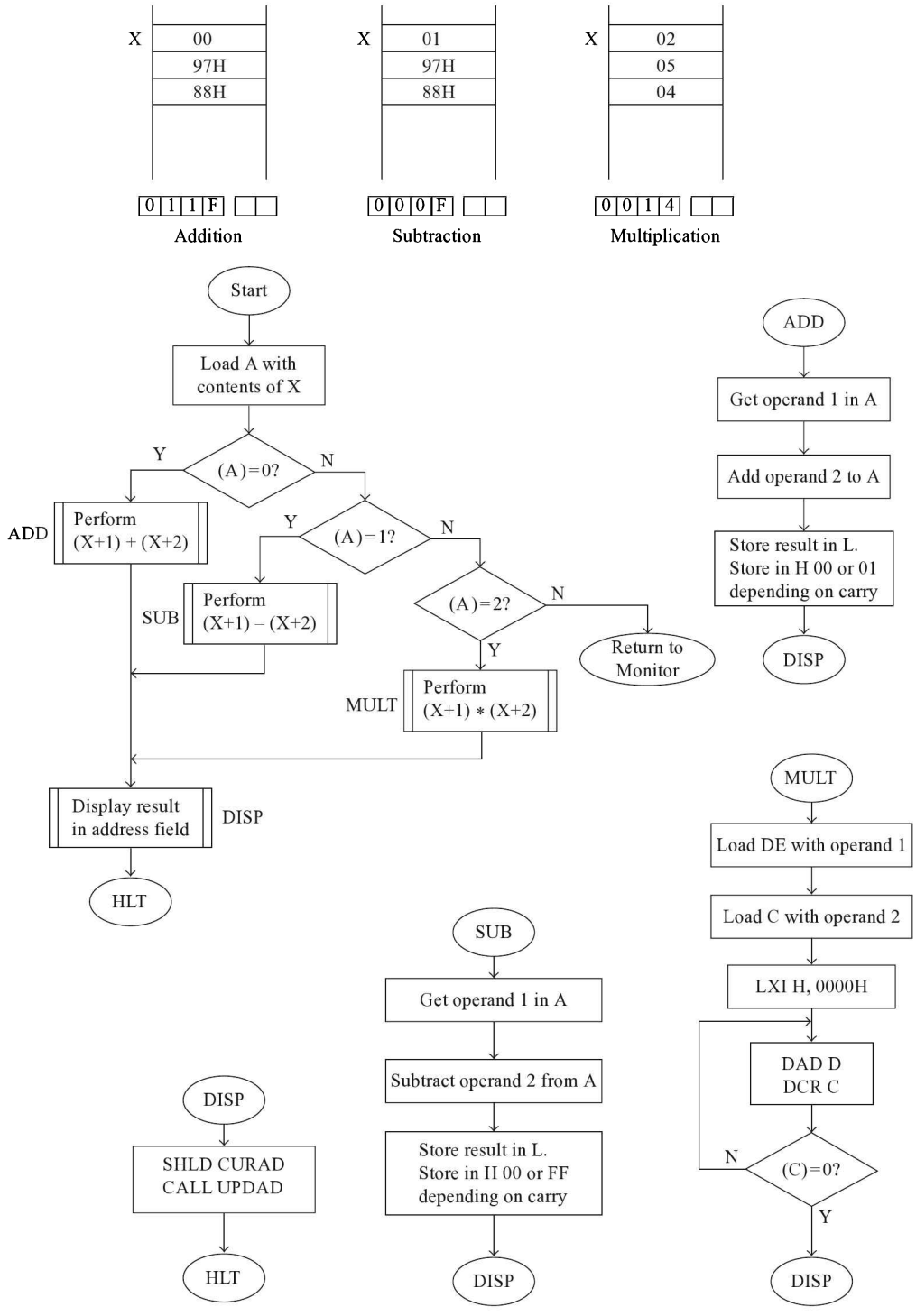


Fig. 17.3 Flowchart to do an operation on two numbers based on the contents of X

```
CURAD: EQU FFF7H
UPDAD: EQU 06BCH

       LXI H, X
       MOV A, M ;;Load A from location X.

       CPI 00H
       JZ ADD   ;;If A value is 00, perform Addition.

       CPI 01H
       JZ SUB   ;;If A value is 01, perform subtraction.

       CPI 02H
       JZ MULT  ;;If A value is 02, perform multiplication.
       RST 1     ;If none of the above, return to monitor.

       ;------ PROGRAM SEGMENT TO PERFORM ADDITION ----------

       ORG C020H
ADD:   INX H
       MOV A, M    ;;Get the first operand in A.

       INX H
       ADD M       ;;Add the second operand to A.
       MOV L, A     ;Store result in L.
       JNC SKIP

       MVI H, 01H;If Cy = 1, load H with 01.
       JMP DISP

SKIP:  MVI H, 00H ;If Cy = 0, load H with 00.
       JMP DISP  ;Jump to DISP, to display the result.

       ;------ PROGRAM SEGMENT TO PERFORM SUBTRACTION ----------

       ORG C040H
SUB:   INX H
       MOV A, M ;;Load A with first operand.

       INX H
       SUB M    ;;Subtract second operand from A.
       MOV L, A  ;Load L with the result.
       JNC SKIP1
       MVI H, FFH  ;Load H with FF, if Cy = 1.
       JMP DISP

SKIP1: MVI H, 00H ;Load H with 00, if Cy = 0.
       JMP DISP   ;Jump to DISP, to display the result.

       ;------ PROGRAM SEGMENT TO PERFORM MULTIPLICATION ----------

       ORG C060H
MULT:  INX H
       MOV E, M
       MVI D, 00H ;;;Load DE with the first operand.

       INX H
       MOV C, M    ;;Load C with the second operand.

       LXI H, 0     ;Initialize HL with 0000H.
```

```
REP:   DAD D        ;Add first operand to HL.
       DCR C        ;Decrement C.
       JNZ REP      ;If nonzero, jump to REP.

       JMP DISP     ;Jump to DISP, to display the result.

       ;------ PROGRAM SEGMENT TO DISPLAY RESULT ----------

       ;At this point, the result of arithmetic operation is in HL.

       ORG C080H
DISP:  SHLD CURAD
       CALL UPDAD ;;Display result in address field.
       HLT
```

■ 17.4 DO AN OPERATION ON TWO BCD NUMBERS BASED ON THE VALUE OF X

Write an 8085 assembly language program to perform an operation on two BCD numbers at X+1 and X+2, based on the contents of X. The operation to be performed is addition, subtraction, or multiplication if the contents of X are 00, 01, or 02, respectively.

Flowchart for solving the problem is shown in Fig. 17.4.

17.4.1 Program to Do an Operation on two BCD Numbers Based on the Contents of X

```
;FILE NAME C:\ALS\BCDFUNC.ASM

  ;8085 ALP TO PERFORM THE FOLLOWING FUNCTIONS BASED ON THE CONTENTS
  ;OF LOCATION X. NEXT 2 LOCATIONS ARE HAVING BCD NUMBERS.
    ;IF CONTENTS OF X IS 00H, PERFORM ADDITION OF NEXT TWO LOCATIONS.
    ;IF CONTENTS IS 01H, PERFORM SUBTRACTION OF NEXT TWO LOCATIONS.
    ;IF CONTENTS IS 02H, PERFORM MULTIPLICATION OF NEXT TWO LOCATIONS.
    ;DISPLAY RESULT IN THE ADDRESS FIELD IN EACH OF THE ABOVE CASES.

      ORG C100H
   X: DB 00H, 97H, 88H

      ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH

       LXI H, X
       MOV A, M ;;Load A with contents of location X

       CPI 00H
       JZ ADD   ;;If A value is 00H, jump to ADD

       CPI 01H
       JZ SUB   ;;If A value is 01H, jump to SUB

       CPI 02H
```

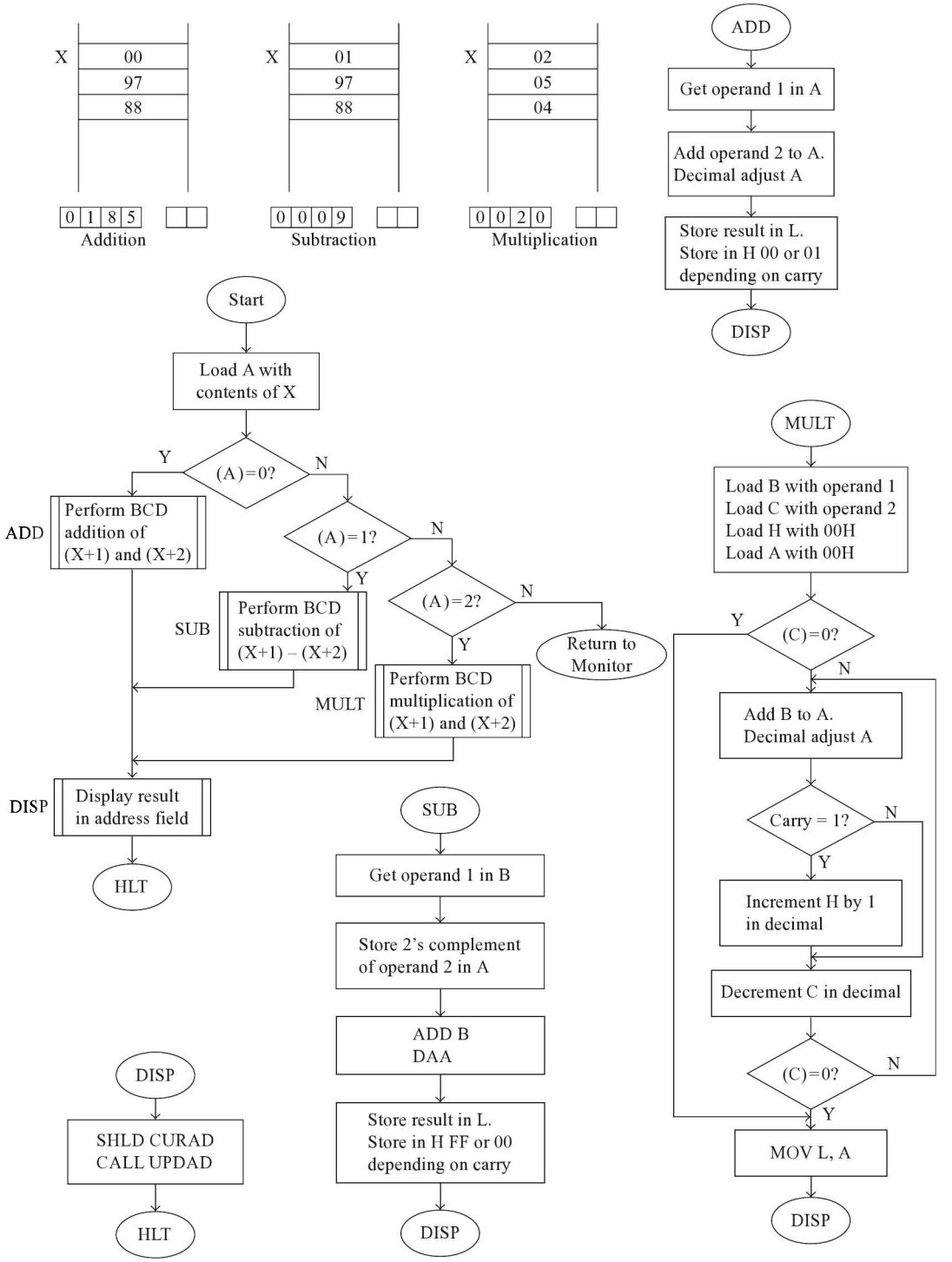


Fig. 17.4 Flowchart to do an operation on two BCD numbers based on the contents of X

```
        JZ MULT  ;;If A value is 02H, jump to MULT
        RST 1

        ;------ PROGRAM SEGMENT TO PERFORM BCD ADDITION ----------

        ORG C020H
ADD:    INX H
        MOV A, M ;;Load A with contents of location X+1

        INX H
        ADD M    ;;Add contents of location X+2
        DAA       ;Decimal adjust

        MOV L, A  ;Load L with result in A

       ;The next 5 instructions load H with 01 or 00 depending on whether
       ;Cy = 1 or 0, and jumps to DISP to display result in the address field.

        JNC SKIP
        MVI H, 01H
        JMP DISP

SKIP:   MVI H,00H
        JMP DISP

        ;------ PROGRAM SEGMENT TO PERFORM BCD SUBTRACTION ----------

     ;The method followed here is as follows. Suppose we want to
     ;perform 85 - 38. We do it as 85 + 62 = 47 with Cy = 1. In this case,
     62 is the 10's complement of 38. Note that when Cy = 1, result ;is
     positive.

     ;As another example, suppose we want to perform 38 - 85. We
     ;should get the correct answer as -47. We do it as 38 + 15 = 53 with
     ;Cy = 0. In this case, 15 is the 10's complement of 85. Note that
     ;when Cy = 0, result is negative, and the result will be in 10's
     ;complement notation. Here, 53 is 10's complement of 47.

       ORG C040H
SUB:    INX H
        MOV B, M  ;;Load B with contents of location X+1

        ;The next 4 instructions result in A having 10's complement
        ;of the number at location X+2

        MVI A, 99H ;Load A with 99
        INX H
        SUB M      ;Perform (99 - contents of X+2) to get 9's complement
        ADI 01H    ;Add 01 to get 10's complement

        ADD B      ;Equivalent to performing (X+1) - (X+2)
        DAA        ;Perform decimal adjustment

        MOV L, A   ;Load L with result in A

     ;The next 5 instructions load H with FF or 00 based on whether Cy value.
     ;is 0 or 1, and then jumps to DISP, to display result in the address field.

        JC SKIP1
        MVI H, FFH
        JMP DISP
```

```
SKIP1: MVI H,00H
       JMP DISP

       ;------ PROGRAM SEGMENT TO PERFORM BCD MULTIPLICATION ----------

       ORG C060H
MULT:  INX H
       MOV B, M ;;Load B with contents of location X+1

       INX H
       MOV C, M ;;Load C with contents of location X+2

       MVI H, 00H;Initialise H with 00. H will finally have
                 ;MS byte of product

       MVI A, 00H;Initialise A. A will finally have LS byte of product.
       CMP C     ;Check if C contents is 00
       JZ EXIT   ;If so, jump to EXIT, load L with A and jump to DISP.

REP:   ADD B     ;Add B contents
       DAA       ;Decimal adjust
       MOV D, A  ;Save result in D temporarily
       JNC NOINRH;If Cy = 0, jump to NOINRH

       ;The next 4 instructions increment H by 1 in decimal

       MOV A, H  ;Load A from H
       ADI 01H   ;Add 01 to A
       DAA       ;Decimal adjust
       MOV H, A  ;Save result in H

    ;The next 4 instructions decrement C by 1 in decimal. For this
    ;purpose, it adds 99, which is 10's complement of 1, and performs
    ;decimal adjustment.

NOINRH:MOV A,C ;Load A from C
       ADI 99H ;Add 99, which is same as subtract 1
       DAA     ;decimal adjust
       MOV C, A;Save result in C

       MOV A, D;Load A from saved value in D
       JNZ REP ;If result of DAA i.e. decrement of C is non zero, jump to REP

       ;When we come out of this loop, H and A will have
       ;MS and LS byte of product respectively

EXIT:  MOV L, A  ;Load L with LS byte of product
       JMP DISP ;Jump to DISP to display product in address field

       ;------ PROGRAM SEGMENT FOR DISPLAY OF RESULT ----------

       ORG C090H
       ;At this point, result of addition, or subtraction, or
       ;multiplication, as the case may be, will be in HL

DISP:  SHLD CURAD
       CALL UPDAD ;Display result in the address field
       HLT
       END
```

■ 17.5 BUBBLE SORT IN ASCENDING/DESCENDING ORDER AS PER CHOICE

Write an 8085 assembly language program to perform sorting in ascending or descending order, based on the contents of location CHOICE, using bubble sort technique. If the contents of CHOICE = 00, sorting should be in ascending order, else in descending order.

Flowchart for solving the problem is shown in Fig. 17.5a.

17.5.1 Program to Perform Bubble Sort in Ascending/Descending Order

```
;FILE NAME C:\ALS\BUBSRT2.ASM

;8085 ALP TO SORT N ONE BYTE BINARY NUMBERS IN ASCENDING/DESCENDING ORDER
;USING BUBBLE SORT. N IS AT LOCATION X, AND THE NUMBERS FROM LOCATION X+1.

;IF CONTENTS OF LOCATION CHOICE = 00, IT IS SORTED IN ASCENDING ORDER,
;ELSE IN DESCENDING ORDER.

      ORG C100H
X:    DB 04H, 33H, 22H, 44H, 11H

      ORG C200H
CHOICE:DB 00H

      ORG C000H

;The next 8 instructions store in location MODIFY the opcode
;for JNC if conents of CHOICE = 00. If contents of CHOICE is <> 00,
;it stores in MODIFY the opcode for JC.

      LDA CHOICE ;Load A from location CHOICE.
      CPI 00
      JZ ASCEND ;;If CHOICE = 00, jump to ASCEND.

      MVI A, D2H  ;Load A with D2, the opcode for JNC.
      STA MODIFY ;If CHOICE <> 00, store code for JNC in MODIFY.
      JMP BEGIN  ;Jump to BEGIN, to sort in descending order.

ASCEND:MVI A, DAH  ;Load A with DA, the opcode for JC.
      STA MODIFY  ;As CHOICE = 00, store code for JC in MODIFY,
                  ;and begin sorting in ascending order.

BEGIN: LXI H, X
      MOV C, M  ;;Load C from location X.
      DCR C      ;Decrement C. C indicates the number of
                 ;passes still to be made.

OUTLOOP:
      MVI E, 01H ;Load E with 1. Contents of E indicates one more
                 ;than the number of exchanges made in a pass.
      MOV B, C   ;Load B from C. B indicates the number of
                 ;comparisons still to be made in a pass.
      INX H      ;Point HL to the first element of the array.
```

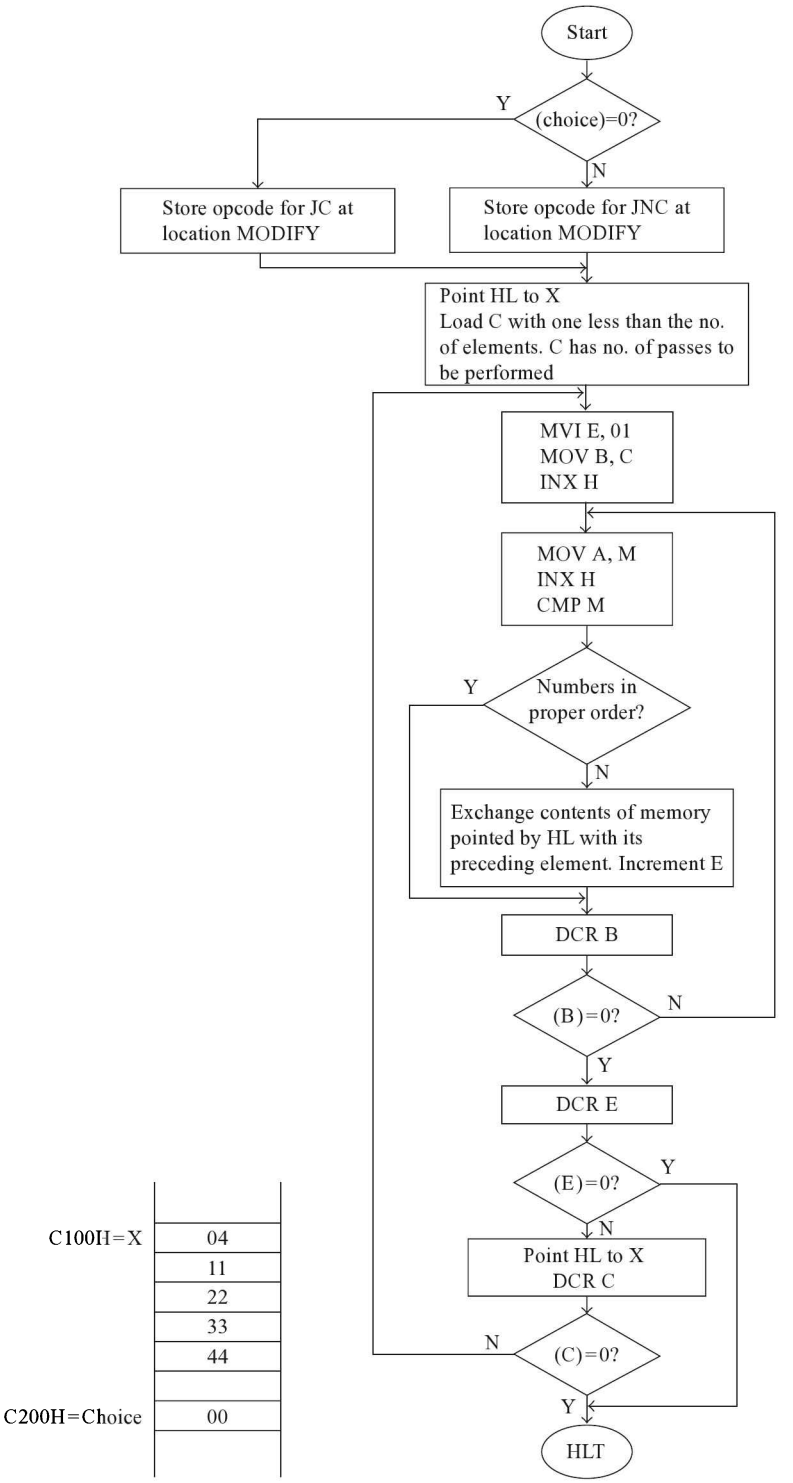


Fig. 17.5a Flowchart to bubble sort in ascending/descending order based on CHOICE

```
INLOOP:
       MOV A, M    ;Load A from memory pointed by HL.
       INX H       ;Point HL to the next element of the array.
       CMP M       ;Compare A (previous element) and next element.

MODIFY:NOP
       DW SKIP     ;;If they are in proper order, jump to SKIP.

  ;The following 6 instructions perform exchange of the bytes
  ;which were not in order, and increments E register.

EXCH:  MOV D, M
       MOV M, A
       DCX H
       MOV M, D
       INX H
       INR E

SKIP:  DCR B       ;Decrement B.
       JNZ INLOOP  ;If nonzero, jump to INLOOP to do next comparison.
       DCR E       ;Decrement E. 00 implies no exchanges were made.
       JZ EXIT     ;So, if 00, jump to EXIT, as sorting is over.

       LXI H, X    ;Point HL to X. From X+1, the array begins.
       DCR C       ;Decrement C.
       JNZ OUTLOOP ;If non zero, jump to OUTLOOP to do next pass.

EXIT: HLT
```

In the above program, it is to be noted that the program itself is modified based on the contents of location CHOICE. However, if we do not want the modification of the program, an alternative program is shown, with flowchart as in Fig. 17.5b.

17.5.2 Alternative Program to Perform Bubble Sort Based on Choice

```
  ;FILE NAME C:\ALS\BUBSRT3.ASM

  ;8085 ALP TO SORT N ONE BYTE BINARY NUMBERS IN ASCENDING/DESCENDING ORDER
  ;USING BUBBLE SORT. N IS AT LOCATION X, AND THE NUMBERS FROM LOCATION X+1.

  ;IF CONTENTS OF LOCATION CHOICE = 00, IT IS SORTED IN ASCENDING ORDER,
  ;ELSE IN DESCENDING ORDER.

       ORG C100H
  X:   DB 04H, 33H, 22H, 44H, 11H

       ORG C200H
CHOICE:DB 00H

       ORG C000H

       LXI H, X
       MOV C, M ;;Load C from location X.
       DCR C     ;Decrement C. C indicates the number of
                 ;passes still to be made.
```

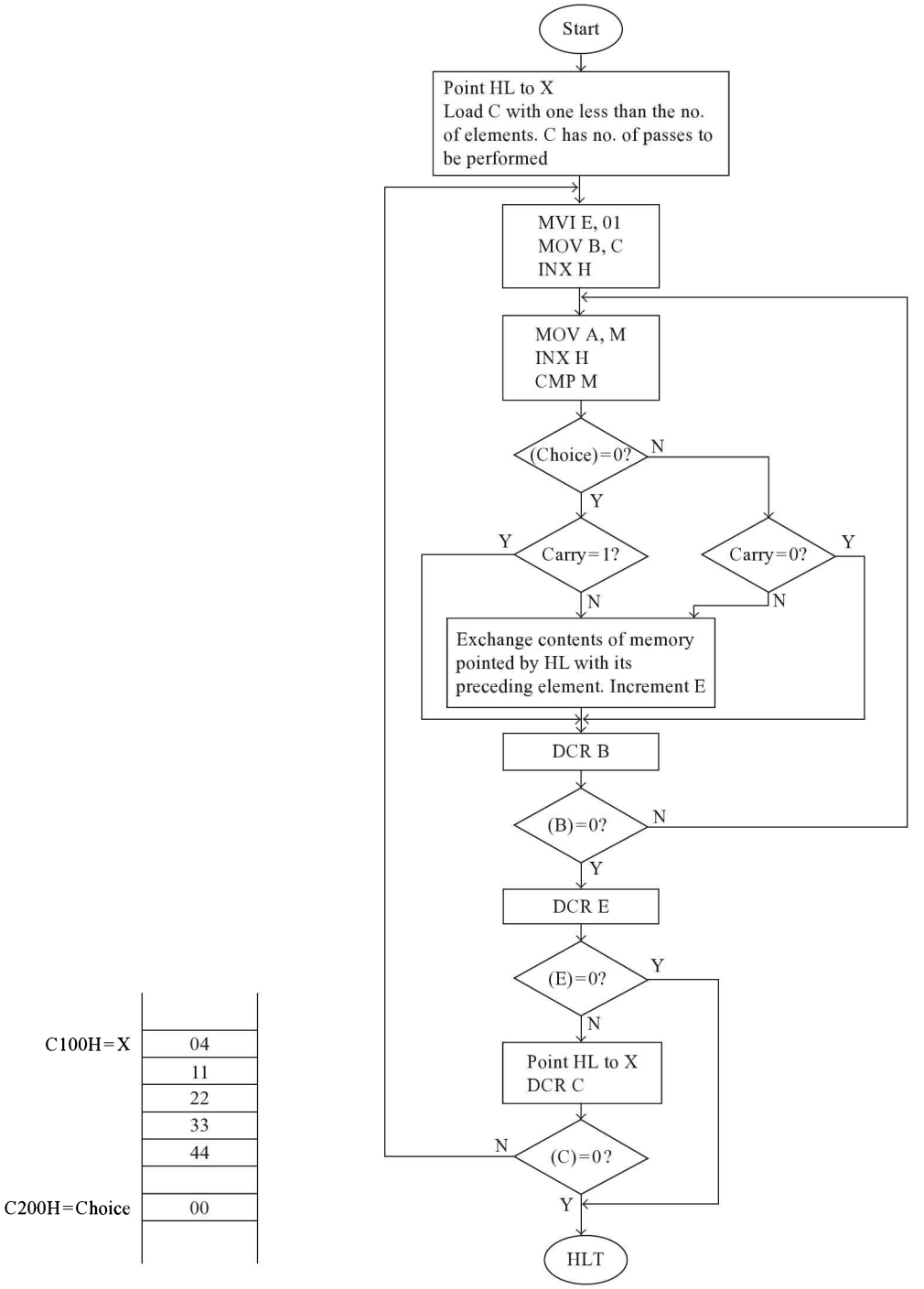


Fig. 17.5b Flowchart showing an alternative method to perform bubble sort based on CHOICE

```
OUTLOOP:
       MVI E, 01H ;Load E with 1. Contents of E indicates one more
                  ;than the number of exchanges made in a pass.
       MOV B, C   ;Load B from C. B indicates the number of
                  ;comparisons still to be made in a pass.
       INX H      ;Point HL to the first element of the array.

INLOOP:
       MOV A, M  ;Load A from memory pointed by HL.
       INX H     ;Point HL to the next element of the array.
       CMP M     ;Compare A (previous element) and next element.

  ;The following 9 instructions perform the following. If
  ;contents of CHOICE = 00, these instructions are equivalent to 'JC
  ;SKIP'. If CHOICE <> 00, they are equivalent to 'JNC SKIP'.

       PUSH PSW
       LDA CHOICE
       CPI 00H
       JZ ASCEND

       POP PSW
       JNC SKIP
       JMP EXCH

ASCEND:POP PSW
       JC SKIP

  ;The following 6 instructions perform exchange of the two bytes
  ;which are out of order, and then increments E by 1.

EXCH:  MOV D, M
       MOV M, A
       DCX H
       MOV M, D
       INX H
       INR E

SKIP:  DCR B        ;Decrement B.
       JNZ INLOOP   ;If non zero, go to INLOOP to do next comparison.

       DCR E        ;Decrement E. 00 implies no exchanges were made.
       JZ EXIT      ;If zero, jump to EXIT, as sorting is over.

       LXI H, X     ;Point HL to X. From X+1 the array starts.
       DCR C        ;Decrement C.
       JNZ OUTLOOP  ;If non zero, jump to OUTLOOP, to do next pass.

EXIT: HLT
```

■ 17.6 SELECTION SORT IN ASCENDING/DESCENDING ORDER AS PER CHOICE

Write an 8085 assembly language program to perform sorting in ascending or descending order based on the contents of location CHOICE, using selection sort technique. If the contents of CHOICE = 00, sorting should be in ascending order, else in descending order.

Flowchart for solving the problem is shown in Fig. 17.6.

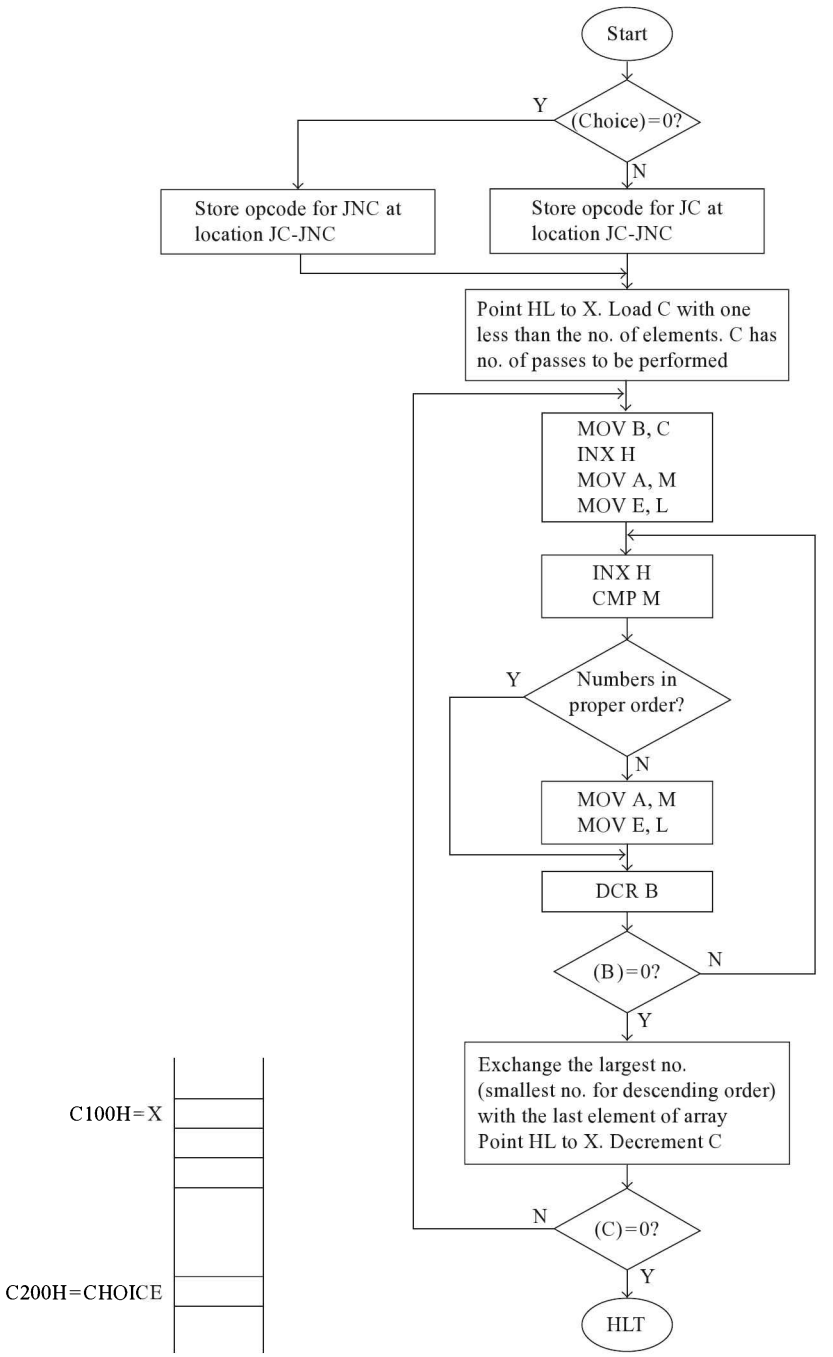


Fig. 17.6 Flowchart to perform selection sort in ascending/descending order based on CHOICE

17.6.1 Program to Perform Selection Sort in Ascending/Descending Order

```
  ;FILE NAME C:\ALS\SELSORT3.ASM

  ;8085 ALP TO SORT N ONE BYTE BINARY NUMBERS IN ASCENDING / DESCENDING ORDER
  ;USING SELECTION SORT. N IS AT LOCATION X, AND THE NUMBERS FROM
  ;LOCATION X+1.

  ;IF CONTENTS OF LOCATION CHOICE = 00, IT IS SORTED IN ASCENDING ORDER,
  ;ELSE IN DESCENDING ORDER.

      ORG C000H
  X:  EQU C100H
CHOICE:EQU C200H

      LDA CHOICE
      CPI 00H
      JZ ASC   ;;;If CHOICE = 0, jump to ASC.

      MVI A, DAH ;If CHOICE <> 0, load A with code for JC.
      STA JC_JNC ;Store code for JC in location JC_JNC.
      JMP SORT   ;Begin sorting in descending order.

ASC:  MVI A, D2H
      STA JC_JNC ;;Store code for JNC in location JC_JNC, and
                  ;begin sorting in ascending order.

SORT: LXI H, X
      MOV C, M   ;;Load C from location X.
      DCR C      ;Decrement C. C contains number of passes still to perform.

OUTLOOP:
      MOV B, C   ;Load B from C. B contains number of
                 ;comparisons still to perform in a pass.

      INX H
      MOV A, M  ;;Load A with the first element of array. Finally A will have
                ;largest (smallest) number for ascending (descending) order.
      MOV E, L   ;Load E with the position of the element.

INLOOP:INX H
      CMP M     ;;Compare A contents with the next element.

JC_JNC:DB        ;It is JC, if CHOICE <> 0 (descending order),
                 ;it is JNC, if CHOICE = 0 (ascending order).
      DW SKIP

PROCEED:
      MOV A, M   ;Load A with larger (smaller) element for
                 ;ascending (descending) order.
      MOV E, L   ;E will have the position of larger (smaller) element.

SKIP: DCR B      ;Decrement B.
      JNZ INLOOP ;If non zero, jump to do next comparison.

      MOV D, M
      MOV M, A
      MOV L, E
```

```
        MOV M, D ;;;;Exchange the largest (smallest) number with
                   ;the last element of the unsorted array for
                   ;sorting in ascending (descending) order.

        LXI H, X     ;Point HL to X. From X+1, the array starts.
        DCR C        ;Decrement C.
        JNZ OUTLOOP  ;If non zero, go to OUTLOOP, to do next pass.
        HLT
```

■ 17.7 ADD CONTENTS OF *N* WORD LOCATIONS

Write an 8085 assembly language program to perform addition of *N* word locations. *N* value is provided in location X. The numbers are stored from word locations X+1. Store the result from location Y. Also display the result in the address field.

Flowchart for solving the problem is shown in Fig. 17.7.

17.7.1 PROGRAM TO ADD THE CONTENTS OF *N* WORD LOCATIONS

```
  ;FILE NAME C:\ALS\ADDNWRD.ASM

  ;8085 ALP TO ADD N 16 BIT NUMBERS STORED FROM LOCATION X+1.
  ;N IS STORED AT LOCATION X. STORE THE RESULT FROM LOCATION Y. ALSO
  ;DISPLAY THE RESULT IN THE ADDRESS FIELD.

       ORG C100H
  X:   DB 03H
       DW 1234H, 5678H, 9ABCH

       ORG C000H

 TEMP: EQU C150H
    Y: EQU C200H
CURAD: EQU FFF7H
UPDAD: EQU 06BCH
DELAY: EQU 04BEH

  ;In this example, we have used the DELAY monitor routine to generate
  ;desired delay. We are required to load DE with the required count value.
  ;The DELAY monitor routine starts at 04BEH in the ALS kit. Some
  ;manufacturers of microprocessor kits do not provide this routine,
  ;as it is simple to be written by the user himself.

       LXI H, 0000H
       SHLD Y+2  ;;Initialize word location Y+2 with 0000H.

       LXI B, X
       LDAX B
       STA TEMP ;;;Save N value in location TEMP.

REP:   INX B
       LDAX B
       MOV E, A ;;;E loaded with LS byte of a 16 bit number.

       INX B
```

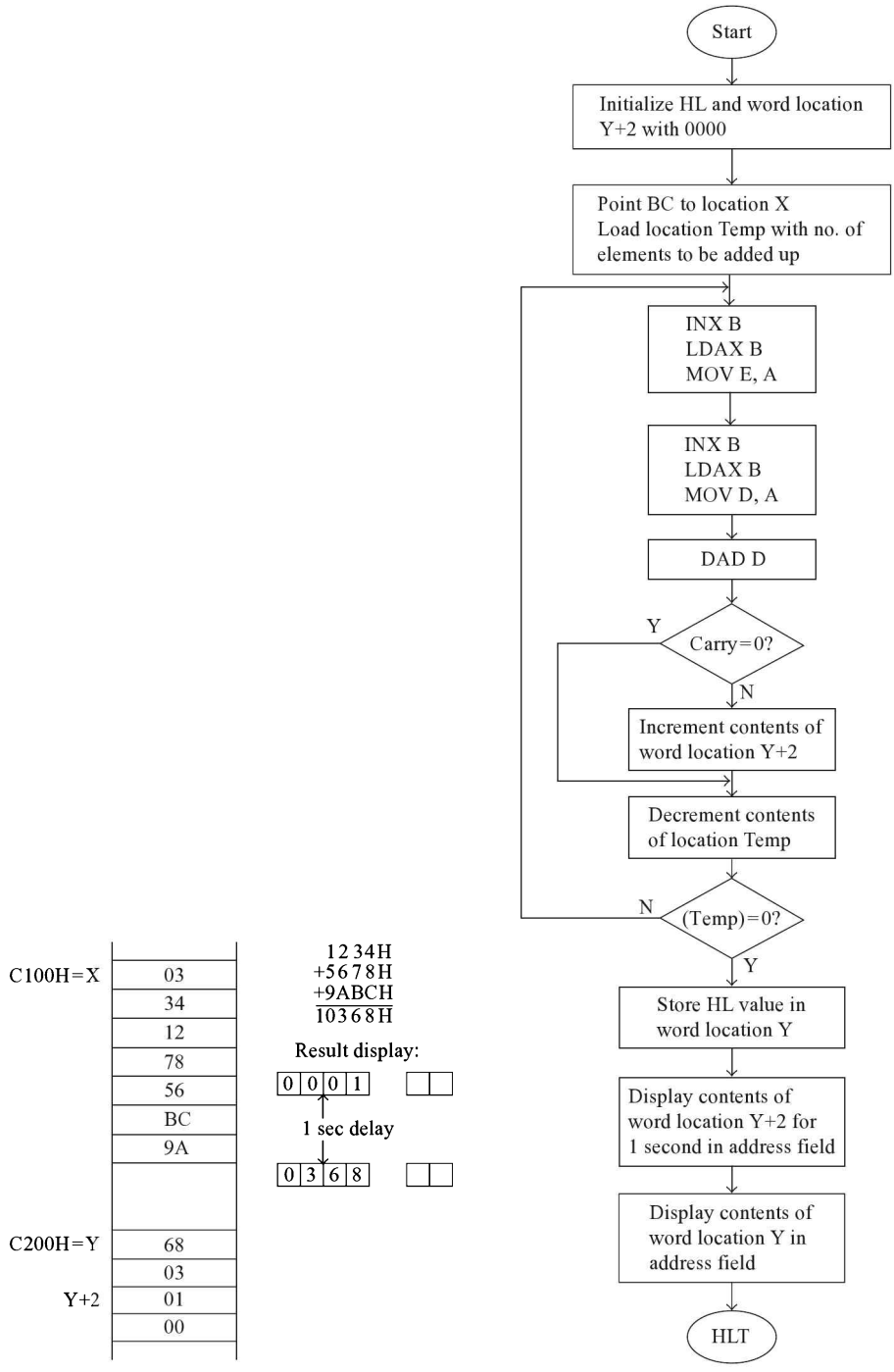


Fig. 17.7 Flowchart to add the contents of *N* word locations

```
        LDAX B
        MOV D, A ;;;D loaded with MS byte of the 16 bit number.

        DAD D       ;Add to HL contents of DE.
        JNC SKIP    ;If Cy = 0, jump to SKIP.
        PUSH H
        LHLD Y+2
        INX H
        SHLD Y+2
        POP H     ;;;;;Else, increment contents of word location Y+2.

SKIP:   LDA TEMP
        DCR A
        STA TEMP    ;;;Decrement count value in TEMP.
        JNZ REP       ;If not zero, jump to REP.

        SHLD Y       ;;Save LS word of sum in word location Y.
        LHLD Y+2
        SHLD CURAD
        CALL UPDAD ;;;Display MS word of sum in address field.

        LXI D, FFFFH
        CALL DELAY
        CALL DELAY ;;;Generate delay of 1 second.

        LHLD Y
        SHLD CURAD
        CALL UPDAD ;;;Display LS word of sum in address field.
        HLT
```

■ 17.8 MULTIPLY TWO 8-BIT NUMBERS (SHIFT AND ADD METHOD)

Write an 8085 assembly language program to perform multiplication of two 8-bit numbers. The numbers are at locations X and Y. Display the 16-bit result in the address field.

Flowchart for solving the problem is shown in Fig. 17.8.

17.8.1 Program to Multiply Two 8-bit Numbers (Shift and Add Method)

```
;FILE NAME C:\ALS\MULT2.ASM

;8085 ALP TO MULTIPLY 2 ONE BYTE BINARY NUMBERS STORED AT
;LOCATIONS X AND Y. DISPLAY THE 16 BIT RESULT IN THE ADDRESS FIELD.

;THIS IS AN EFFICIENT PROGRAM USING SHIFT AND ADD METHOD.

    ORG C100H
X:  DB 0AH

    ORG C200H
Y:  DB 05H

    ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH
```

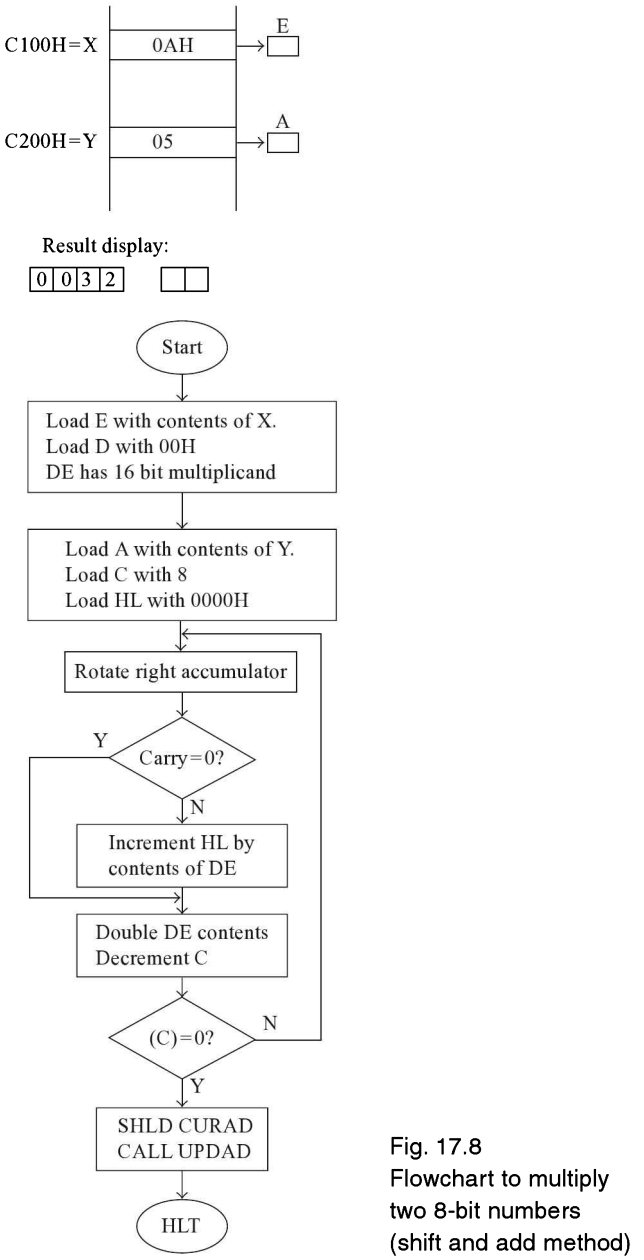


Fig. 17.8
Flowchart to multiply two 8-bit numbers (shift and add method)

```
LXI  H, X
MOV  E, M
MVI  D, 00H  ;Load DE with the number at location X.
LDA  Y       ;Load A with the number at location Y.
MVI  C, 08   ;Load C with 8. It indicates the number of
             ;shift and add to be still performed.
LXI  H, 0    ;Initialize HL. HL will finally have the product.
```

```
  ;The next 8 instructions perform the following. It checks a
  ;bit of A, starting with the LS bit. If the bit is 0, nothing is
  ;added to HL value. If the bit is 1, DE contents is added to HL. Then
  ;immaterial of the value of the bit, DE is doubled. Counter C is
  ;decremented. If not zero, the loop is repeated.

AGAIN: RRC          ;Check a bit of A, starting with the LS bit.
       JNC SKIP     ;If Cy = 0, jump to SKIP.
       DAD D        ;If Cy = 1, increment HL by the contents of DE.

SKIP:  XCHG
       DAD H
       XCHG      ;;;Double DE contents.

       DCR C        ;Decrement C.
       JNZ AGAIN    ;If not zero, jump to AGAIN.

       ;When we come out of this loop, HL will have the product.

       SHLD CURAD
       CALL UPDAD ;;Display the product in the address field.
       HLT
```

■ 17.9 MULTIPLY TWO 2-DIGIT BCD NUMBERS

Write an 8085 assembly language program to perform multiplication of two 2-digit BCD numbers. The numbers are at locations X and X+1. Display the four-digit BCD result in the address field.

Flowchart for solving the problem is shown in Fig. 17.9.

17.9.1 PROGRAM TO MULTIPLY TWO 2-DIGIT BCD NUMBERS

```
;FILE NAME C:\ALS\MUL_BCD.ASM

  ;8085 ALP TO MULTIPLY TWO 2-DIGIT BCD NUMBERS STORED AT X
  ;AND X+1. DISPLAY THE 4-DIGIT BCD RESULT IN THE ADDRESS FIELD. IT
  ;USES REPEATED ADDITION.
       ORG C100H
  X:   DB 04H, 05H

       ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH

       LXI H, X
       MOV B, M ;;Load B from contents of location X.

       INX H
       MOV C, M   ;;Load C from contents of location X+1.

       MVI E, 00H ;Initialize E with 00. After every addition, E will be
                  ;incremented in decimal notation till E becomes equal to C.

       MVI H,00H  ;Initialize H with 00. H will finally have
                  ;MS byte of BCD product.

       MVI A, 00H ;Initialise A with 00. A will finally have
                  ;LS byte of BCD product.
```

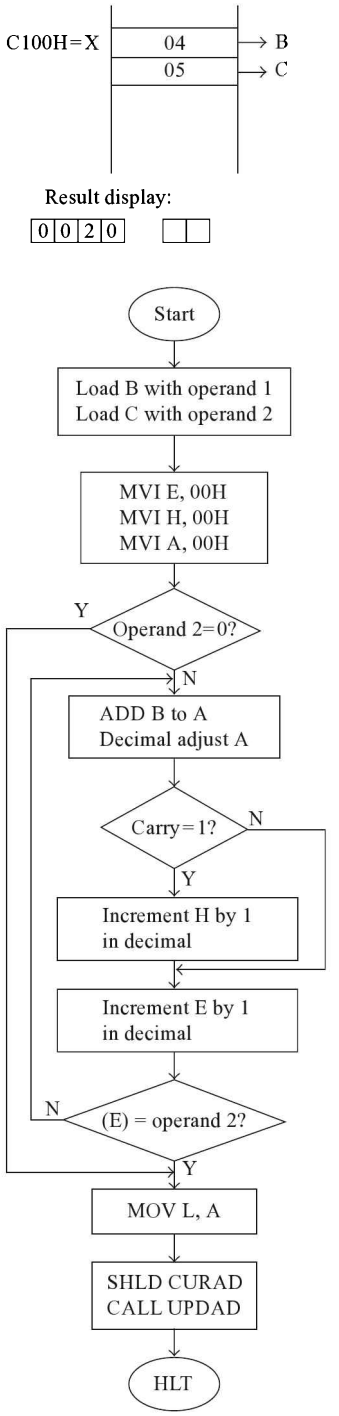


Fig. 17.9 Flowchart to multiply two 2-digit BCD numbers

```
        CMP C
        JZ EXIT    ;;If contents of C is zero, jump to EXIT.
REP:    ADD B
        DAA        ;;Add to A contents of B in decimal notation.
        MOV D, A   ;Save A value in D register.
        JNC NOINRH ;If Cy = 0, jump to NOINRH.
        MOV A, H
        ADI 01H
        DAA
        MOV H, A ;;;;If Cy = 1, increment H in decimal.
NOINRH:MOV A, E
        ADI 01H
        DAA
        MOV E,A ;;;;Increment E in decimal.
        CMP C      ;Compare E and C contents.
        MOV A, D   ;Reload A from D register.
        JNZ REP    ;If E and C are not same, jump to REP.
    ;When we come out of this loop, the product will be in H and A.
EXIT: MOV L,A      ;Load L with LS byte of product.
      SHLD CURAD
      CALL UPDAD;;Display BCD product in the address field.
      HLT
```

■ 17.10 MULTIPLY TWO 16-BIT BINARY NUMBERS

Write an 8085 assembly language program to perform multiplication of two 16-bit binary numbers. The numbers are at locations X, X+1 and Y, Y+1. Store the 32-bit result in four locations starting from location Z.

Flowchart for solving the problem is shown in Fig. 17.10.

17.10.1 Program to Multiply two 16-bit Binary Numbers

```
;FILE NAME C:\ALS\MUL16.ASM
;8085 ALP TO MULTIPLY TWO 16 BIT BINARY NUMBERS STORED AT
;LOCATIONS X, X+1 AND Y, Y+1. STORE THE 32 BIT RESULT IN 4 LOCATIONS
;STARTING AT Z. IT USES REPEATED ADDITION.
        ORG C100H
  X:    DW 0004H
        ORG C200H
  Y:    DW 0005H
  Z:    EQU C300H
        ORG C000H
        LHLD X
        XCHG       ;;Load DE with contents of word location X.
```

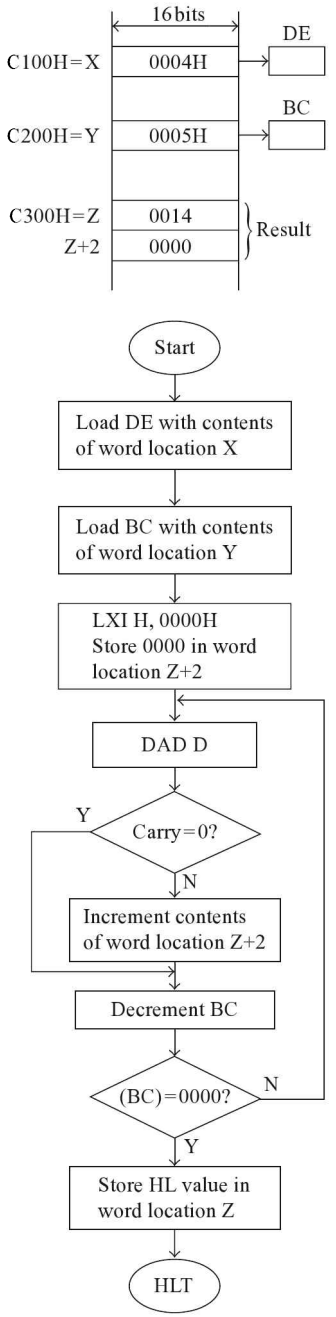


Fig. 17.10
Flowchart to multiply two 16-bit binary numbers

```
LHLD Y
PUSH H
POP B     ;;;Load BC with contents of word location Y.

LXI H, 0    ;Load HL with 0000H.
SHLD Z+2    ;Store the value 0000H in word location Z+2.
```

```
REP:   DAD D       ;Add the first number to HL.
       JNC NOINCR  ;If Cy = 0, jump to NOINCR.

       PUSH H
       LHLD Z+2
       INX H
       SHLD Z+2
       POP H     ;;;;;If Cy = 1, increment word location Z+2 contents.

NOINCR:DCX B         ;Decrement BC contents.
       MOV A, B
       ORA C
       JNZ REP   ;;;If non zero, jump to REP.

    ;When we come out of this loop, word location Z+2 will have
    ;MS word of result, and HL will have LS word of result.

       SHLD Z ;Store LS word of result in word location Z.
       HLT
```

1. Write an 8085 assembly language program to exchange a block of words (16 bits) of data starting at locations X and Y. Size in words of the block is provided in location *N*.
2. Write an 8085 assembly language program to search for a given 8-bit data in an array of sorted bytes, using Binary search algorithm. The number of elements is provided in location X, the element to be searched is provided in location X+1, and the array of sorted bytes starts at location X+2. If search is successful, store FF in location *Y* and store the 16-bit address of the position the element was found in word location Y+1. If search is unsuccessful, store 00 in location **Y** and store 0000H in word location Y+1.
3. Write an 8085 assembly language program to compute the HCF of two 16-bit numbers stored in word locations X and X+2. Store the result in word location X+4.
4. Write an 8085 assembly language program to compute the LCM of two 16-bit numbers stored in word locations X and X+2. Store the result in word locations X+4 and X+6.
5. Write an 8085 assembly language program to compute the HCF of two 4-digit BCD numbers stored in word locations X and X+2. Store the result in word location X+4
6. Write an 8085 assembly language program to compute the LCM ot two 4-digit BCD numbers stored in word locations X and X+2. Store the result in word locations X+4 and X+6.
7. Write an 8085 assembly language program to convert a 32 bit binary number stored in locations X to X+3 to equivalent BCD number. Store the result in locations X+4 to X+8.
8. Write an 8085 assembly language program to convert a 8-digit BCD number stored in locations X to X+3 to equivalent binary number. Store the result in locations X+4 to X+7.
9. Write an 8085 assembly language program to sort a given set of 16-bit numbers in ascending order using Bubble sort algorithm. The number of elements to be sorted is provided in location X. The elements are in word locations starting from X+1.
10. Write an 8085 assembly language program to sort a given set of 16-bit numbers in descending order using Selection sort algorithm. The number of elements to be sorted is provided in location X. The elements are in word locations starting from X+1.

III

Programmable and Non-Programmable I/O Ports

Chapter Heads

18 Interrupts in 8085
19 8212 Non-Programmable 8-Bit I/O Port
20 8255 Programmable Peripheral Interface Chip
21 Programs Using Interface Modules

INTRODUCTION

In this part, we will deal in detail with 8255 PPI, which is a very popular chip that is used for interfacing I/O devices. Theoretically, any I/O device could be interfaced with the microprocessor using the 8255. This part comprises of Chaps. 18 to 21. Chapters 18 and 19 deal with the interrupts in the 8085 microprocessor and the non-programmable chip, Intel 8212, respectively. Chapters 20 and 21, which form the latter portion of this part deal with the programmable I/O port, 8255 and programs using interface modules.

18 Interrupts in 8085

- Data transfer schemes
  - *Basic or simple data transfer*
  - *Status check data transfer*
  - *Interrupt-driven data transfer*
- General discussion about 8085 interrupts
- EI and DI instructions
  - *Reset_in* and Reset_out pins*
- INTR and INTA* pins
  - *Action taken by 8085 when INTR pin is activated*
  - *Alternative action by 8085 when INTR pin is activated*
- RST5.5 and RST6.5 pins
- RST7.5 pin
- Trap interrupt pin
  - *Action taken when 8085 is interrupted due to a vector interrupt*
- Execution of 'DAD rp' instruction
- SIM and RIM instructions
  - *Need for masking*
  - *SIM instruction*
  - *RIM instruction*
- HLT instruction
- Programs using interrupts
  - *Program for simulation of throwing a die*
  - *Program for simulating a stopwatch*
  - *Program to find Square of a number using look up table*
  - *Program for decimal down counter*
  - *Program for decimal down counter (alternative)*
  - *Program for adding 2 numbers input from keyboard*
  - *Program for adding 4 hex digits of a 16-bit number*
- Questions

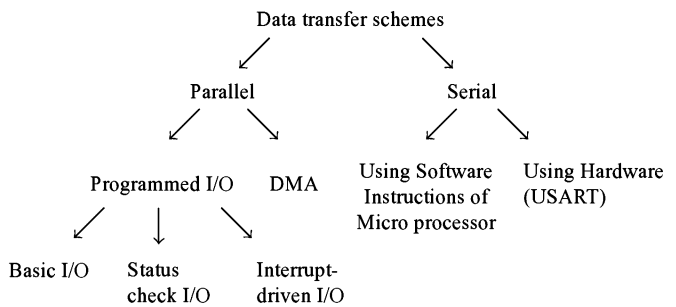
So far, we have discussed only 28 pins out of the 40 pins present in 8085. In this chapter, we discuss ten more pins of 8085. These are mostly related to interrupts in 8085. Also, only 242 instructions out of 246 instructions of 8085 have been discussed so far. In this chapter, the remaining four instructions, which are related to interrupts are also discussed.

■ 18.1 DATA TRANSFER SCHEMES

When the 8085 is executing a program, it can get interrupted half way through the program by an I/O device. An I/O device interrupts the working of the processor, because it may want to urgently communicate with the processor. It may want to send some information to the processor, or receive some information from the processor.

A microprocessor does not directly communicate with an I/O device. It communicates with an I/O device via an I/O port. Data transfer can be in parallel or serial form. Parallel data transfer is possible using programmed I/O or Direct Memory Access (DMA). Serial data transfer and DMA data transfer are explained in later chapters. There are three different ways a microprocessor can communicate with an I/O port for parallel data transfer with programmed I/O. They are:

1. Basic or simple data transfer;
2. Status check data transfer;
3. Interrupt-driven data transfer.



18.1.1 Basic or Simple Data Transfer

This is the simplest of the data transfer schemes. This method is useful when we have accurate knowledge of the I/O device timing characteristics. When we know that the device is ready for data transfer, we execute IN or OUT instruction, depending on the required direction of data transfer. This is the case when the I/O port is connected in the system as an I/O-mapped I/O port. If the port is connected as a memory-mapped I/O port, 'MOV M, A', 'MOV A, M', or any other memory reference instruction is used depending on the direction of data transfer.

As an example, let us say we have a hypothetical multiplier chip. Registers R0 and R1 are used to load the two 8-bit numbers to be multiplied. The 16-bit product will be available in R2H and R2L when the multiplication is over. Address pins A1 and A0 select a register as shown in the following.

A1	*A0*	*Selected register*
0	0	R0
0	1	R1
1	0	R2L (LS byte of product)
1	1	R2H (MS byte of product)

Let us say the chip is connected as I/O-mapped I/O, as shown in Fig. 18.1. As per the chip select circuit, the address for R0, R1, R2L, and R2H is 40H, 41H, 42H, and 43H, respectively. Let us say the data sheet for the multiplier specifies a maximum of 50 μs for completing the multiplication.

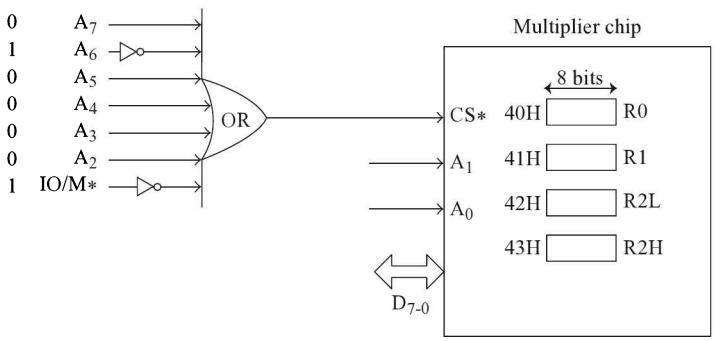


Fig. 18.1
Simple I/O data transfer

With this set up, the following program segment performs multiplication of 05H and 08H, and the result will be stored in the BC register pair.

```
MVI A, 05H
OUT 40H     ;Load R0 with 05H
MVI A, 08H
OUT 41H     ;Load R1 with 08H
CALL DELAY  ;Generate a delay of 50 micro seconds
;The subroutine is not shown for simplicity
IN 42H
MOV C, A    ;Store LS byte of product in C register
IN 43H
MOV B, A    ;Store MS byte of product in B register
```

In the previous example, even if the multiplication is over in 20 μs, we are required to wait for 50 μs, as per the data sheet. The advantage of simple I/O is its simplicity, but its disadvantage is that it is not very efficient. We come across a number of examples later in the text for this type of data transfer. The flowchart for simple I/O is illustrated in Fig. 18.2.

18.1.2 Status Check Data Transfer

This is more complex than simple data transfer. This method is used when we do not have accurate knowledge of the I/O device timing characteristics. The processor should get status information about the readiness of the I/O device for data transfer. Generally, the processor will be in a loop checking for the readiness of the device. The moment the device is ready, it comes out of the loop, and executes IN or OUT instruction depending on the requirement. In this case, the processor has simply wasted its time in a loop till the device got ready.

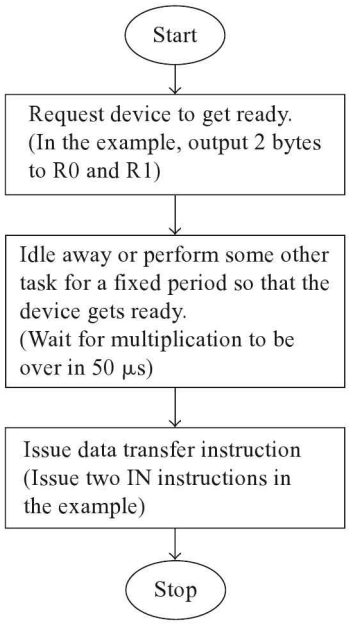


Fig. 18.2
Flowchart for simple I/O

As an example, let us once again consider the hypothetical multiplier chip. Let the chip be provided with command and status registers also. Let us say when the MS bit of the command register is set to 1, it is a command to start the multiplication of the two numbers in R0 and R1. Let us say the LS bit of the status register is set to 1 when the product is available in R2H and R2L.

Address pins A2, A1, and A0 select a register as shown in the following.

A2	*A1*	*A0*	*Selected register*
0	0	0	R0
0	0	1	R1
0	1	0	R2L (LS byte of product)
0	1	1	R2H (MS byte of product)
1	0	0	Command register
1	0	1	Status register

Let us say the chip is connected as I/O-mapped I/O, as shown in Fig. 18.3. As per the chip select circuit, the address for R0, R1, R2L, R2H, command register and status register is 40H, 41H, 42H, 43H, 44H, and 45H, respectively.

In this case, the microprocessor sends the two bytes to be multiplied to the registers R0 and R1. Then a command is issued to start the multiplication process. This is done by setting the MS bit of the command register. Then the status register is continuously monitored for the completion of the multiplication operation. This is done by checking the LS bit of status register. When the LS bit of status register becomes 1, the microprocessor reads the result.

With this set up, the following program segment performs multiplication of 05H and 08H, and the result will be stored in the BC register pair.

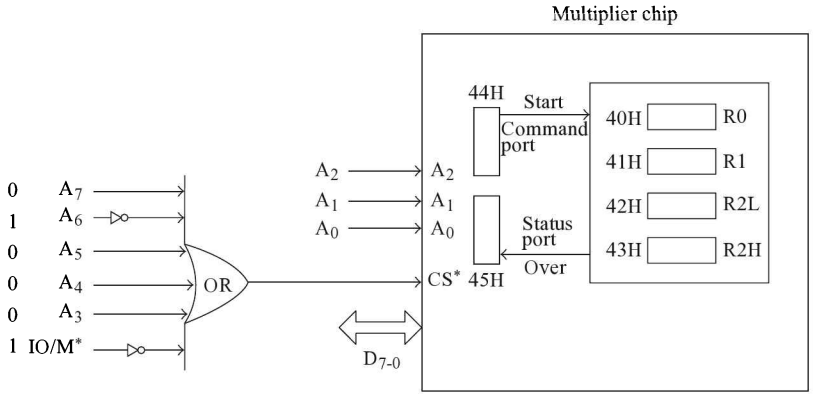


Fig. 18.3
Status check data transfer

```
        MVI A, 05H
        OUT 40H     ;Load R0 with 05H
        MVI A, 08H
        OUT 41H     ;Load R1 with 08H
        MVI A, 10000000B
        OUT 44H     ;Issue Start multiplication command

WAIT:   IN 45H
        RRC
        JNC WAIT    ;Be in wait loop till LS bit of Status
                    ;register becomes 1

        IN 42H
        MOV C, A    ;Store LS byte of product in C register
        IN 43H
        MOV B, A    ;Store MS byte of product in B register
```

In status check I/O, as soon as the device is ready, the data transfer is performed. Status check I/O is more efficient than simple I/O, but is more complex. This method of data transfer is also known as *handshake data transfer*.

We come across a number of examples later in the text for this type of data transfer. The flowchart for status check I/O is illustrated in Fig. 18.4.

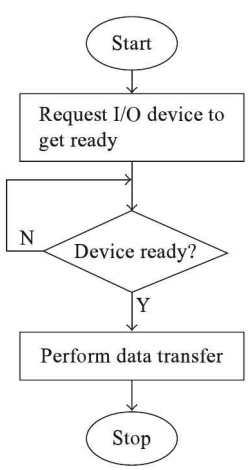


Fig. 18.4
Flowchart for status check I/O

18.1.3 Interrupt-Driven Data Transfer

This method is used when we do not have accurate knowledge of the I/O device timing characteristics, except that it takes quite a long time for the device to get ready. If we resort to status check data transfer, the processor will have to waste a long time in the loop for the device to get ready. To avoid this problem, interrupt-driven data transfer can be used. In this case, the processor will go ahead with its required work, and whenever the device gets ready for data transfer, the corresponding I/O port will send a interrupt request signal to the processor. The interrupt request may arrive even half way through an instruction execution. Then the processor will complete the instruction. After this, the processor will perform data transfer with the I/O device using IN or OUT instruction. Then it resumes the execution of the interrupted program. Flowchart for interrupt-driven data transfer is provided in Fig. 18.5a.

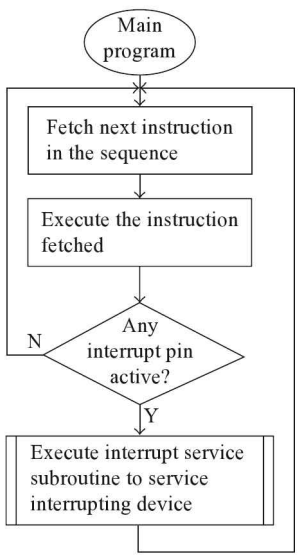


Fig. 18.5a
Flowchart for interrupt-driven data transfer

This method is obviously the most complex of the three types of data transfer. But the advantage now is that the processor is not wasting its time in a loop checking for the readiness of the I/O device.

For example, if the 8085 wants to read from a keyboard, one way is to continuously scan the keyboard looking for a pressed key. This is the status check method. It may so happen that once in a second or so, it finds that a key is pressed. The 8085 is in a loop for this amount of time just waiting for a key depression. In this time, the 8085 could have executed about 500,000 instructions, assuming an average execution time of 2 μs per instruction! In interrupt-driven data transfer, the 8085 would execute about 500,000 useful instructions, by which time the user presses a key on the keyboard. This causes an interrupt signal to be sent to the 8085 by the I/O port. Then the 8085 reads from the port attached to the keyboard, after which it goes ahead with the program.

In the remaining portion of this chapter, we mainly discuss the interrupt pins of 8085, interrupt-related instructions, and some programs that use interrupts. These programs clearly describe the interrupt-driven data transfer.

■ 18.2 GENERAL DISCUSSION ABOUT 8085 INTERRUPTS

Interrupt pins of 8085 are used by I/O devices to initiate transfer of data to or from 8085, without wasting much of the CPU time. As seen previously, this is very useful, when the timing characteristics of the I/O device is not well known, and it takes a long time for the device to get ready for data transfer. There are five interrupt pins on 8085, as shown in Fig. 18.5b. They are input pins of 8085. They are TRAP, RST7.5, RST6.5, RST5.5, and INTR.

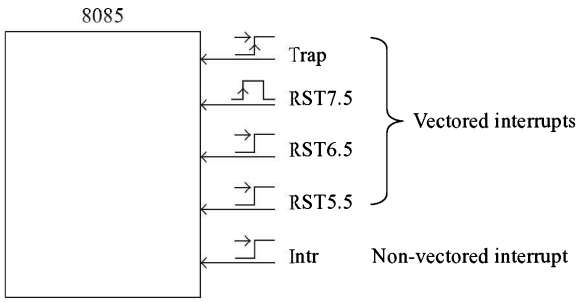


Fig. 18.5b
Interrupt pins of 8085

Note that RST7, RST6, and RST5 are instructions of 8085, whereas RST7.5, RST6.5, and RST5.5 are interrupt pins of 8085. An interrupt pin can be activated by an I/O port even half way through an instruction, without having any relation to a clock signal. Thus the interrupts are asynchronous in nature. The 8085 checks all these interrupt pins at about the end of every instruction. Specifically, they are checked in the penultimate clock cycle of the last machine cycle of an instruction. However, the interrupt pins are not checked at the end of an instruction, if the instruction belongs to the branch group, JMP, CALL, RET, etc. In such cases, the interrupt requests are sensed after a lapse of about 15 clock cycles. At this point, if several interrupts are active simultaneously, the 8085 services them as per the priority shown in the table that follows.

Pin	*Priority*	*Triggering*
TRAP	Highest	Both edge & level
RST7.5	Second	Edge
RST6.5	Third	Level
RST5.5	Fourth	Level
INTR	Lowest	Level

When an I/O port activates one of these pins, the 8085 gets interrupted if each of the following conditions are satisfied.

- The interrupt system is enabled by setting the interrupt enable (IE) flip-flop of 8085. This condition is not applicable to TRAP. The method of setting the interrupt enable flag is discussed later.
- The interrupt pin has not been masked. This condition is not applicable to TRAP and INTR. The method of masking or unmasking an interrupt pin is discussed later.
- Higher priority interrupts are not active at the same time.

The following is the broad action taken by the 8085 when it gets interrupted. Finer details are given later in the chapter.

- It completes the execution of the current instruction.
- Stores on the stack top the address of the next instruction, called Return address, which is present in the program counter.
- Program branches to a subroutine, whose execution satisfies the I/O device that interrupted the 8085. This subroutine is thus named as an interrupt service subroutine or ISS. Details of ISS address are provided later in the chapter.
- Normally an ISS consists of three portions. The first portion is used for saving on the stack all the register values of 8085 that are going to be affected by the execution of the ISS. Thus, it uses a number of PUSH instructions. The next portion will have instructions to achieve the actual purpose of the ISS. If the purpose is data transfer, we will have IN or OUT instruction in this part. However, if the I/O port is connected as memory-mapped I/O, a memory reference instruction will be used for data transfer with the I/O port. The last part is used for restoring the original values to various registers that were affected by the second part of ISS. As such, it contains a number of POP instructions. The ISS then ends with EI and RET instructions. The purpose of EI instruction will be discussed later. The typical layout of an ISS is provided in Fig. 18.6.
- After completely executing the ISS, the control returns to the program that was interrupted. This is done by popping the top of stack information to the PC, using RET instruction at the end of ISS.

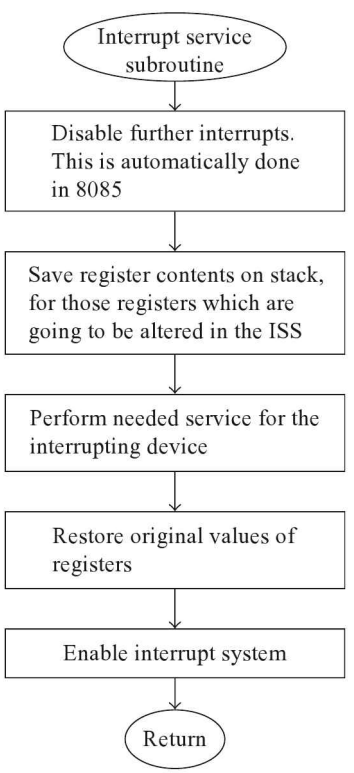


Fig. 18.6
Typical layout of an ISS

■ 18.3 EI AND DI INSTRUCTIONS

In the previous section, a broad outline was provided regarding the action to be taken by the 8085 when an I/O port activated an interrupt pin. The finer details are discussed in the rest of the chapter.

The 8085 can be thought of as having five internal interrupt signals that correspond to the five external interrupt pins. Only when the internal interrupt signal is activated, the 8085 gets interrupted, provided higher priority internal interrupt signals are not active at the same time. The 8085 checks all these internal interrupt signals in the penultimate clock cycle of the last machine cycle of an instruction.

There is a flip-flop in 8085 called IE flip-flop. Here IE stands for Interrupt Enable. Whenever this flip-flop is reset to the 0 state, 8085 interrupt system is disabled. That is, even if an external interrupt pin is activated, the corresponding internal interrupt signal is not activated. This can be seen from Fig. 18.7, which provides the internal architecture of 8085 interrupt structure. From this figure it can also be noticed that this flip-flop state has nothing to do with the internal interrupt signal corresponding to TRAP. TRAP is a non-maskable interrupt. It means whenever the external TRAP pin is activated, the corresponding internal interrupt signal is always activated. Further, TRAP being the highest priority interrupt, it always interrupts the 8085.

The IE flip-flop is reset to the 0 state by the following three conditions as shown in Fig. 18.7.

1. Execution of DI instruction;
2. Recognition of an interrupt request;
3. Resetting of 8085.

The DI instruction stands for 'disable interrupts'. It is an 1-byte instruction. When this instruction is executed, the IE flip-flop is reset. This disables the 8085 interrupt system except for the TRAP pin.

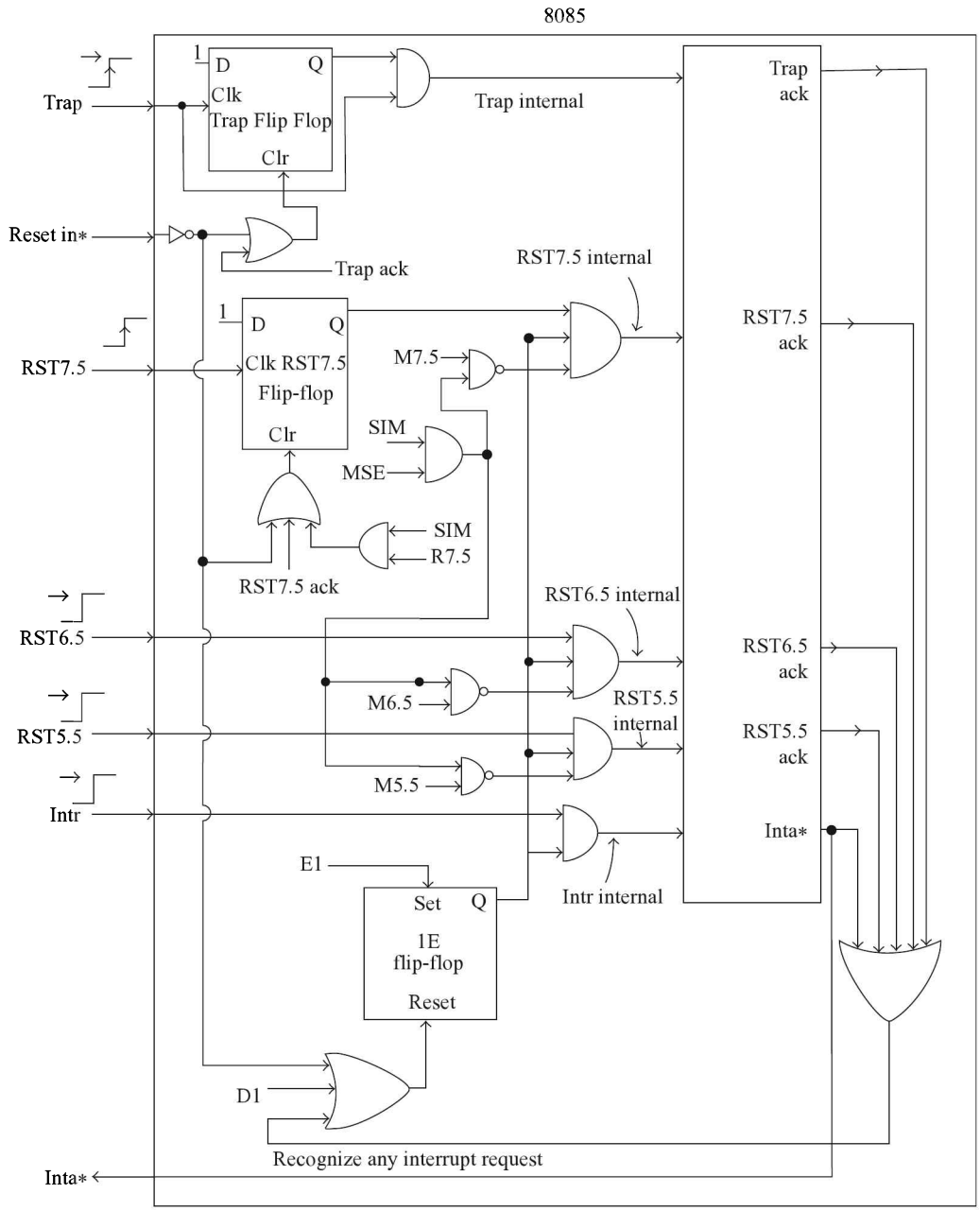
When the 8085 recognizes an interrupt, it branches to an ISS. By recognition we mean that, an internal interrupt signal is activated, and higher priority internal interrupt signals are not active. In the ISS, the 8085 would not like to be interrupted again. That is the reason, as to why the 8085 will place itself in DI state, when it recognizes an interrupt. However, TRAP can still interrupt an ISS. Thus, when the control is transferred to an ISS, interrupt system is disabled automatically. Accordingly, there is no need for the programmer to write a DI instruction at the beginning of an ISS. Whether the programmer writes a DI instruction at the beginning of an ISS or not, interrupt system remains disabled, except for TRAP.

18.3.1 RESET_IN* AND RESET_OUT PINS

Intel 8085 has a RESET_IN* pin. This is an active low input pin. The 8085 is reset by placing a logic 0 on this pin for atleast 0.5 μs, after power is supplied to V_{cc} pin of 8085. However, in practice the RESET_IN* is placed in logic 0 state for atleast a few milliseconds. A typical reset circuit, used in ALS 8085 kit, is shown in Fig. 18.8.

The moment power supply is switched on, V_{cc} pin gets +5 V power. However, the RESET_IN* pin remains in the logic 0 state for a time dependent on the RC time constant. In the circuit in Fig. 18.8, the RC time constant is about 50 μs. Thus, the RESET_IN* pin will be in logic 0 for a time longer than the minimum required 0.5 μs.

When the system is already powered and running, sometimes we need to reset the 8085. There is no need to switch off, and then switch on, to reset the 8085. There is a manual reset switch as shown in Fig. 18.8. When we momentarily press and release it, the RESET_IN* pin goes to logic 0, and then



SIM, EI, and DI signals activated by the control unit when the 8085 executes SIM, EI, and DI instructions, respectively. MSE, M7.5, M6.5, M5.5 are bit 4, bit 3, bit 2, and bit 1 of the accumulator, respectively. R7.5 is bit 4 of the accumulator.

Fig. 18.7 Architecture of 8085 interrupt structure

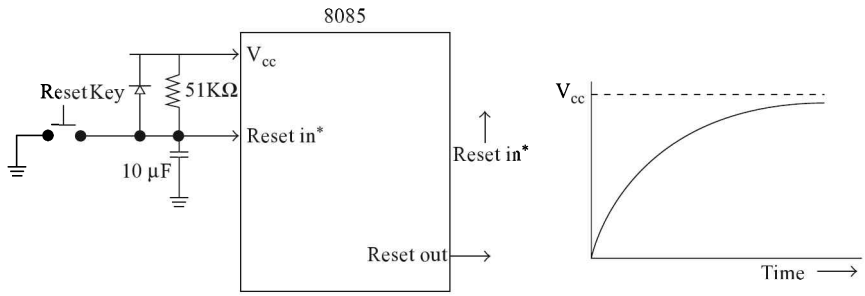


Fig. 18.8
Reset circuit for 8085

comes back to logic 1 state after a time dependent on the RC constant. In this case, if the RESET_IN* pin is kept at logic 0 for atleast three clock cycles (1 μs) the 8085 will be reset.

Some of the important actions performed by the 8085 when it is reset are as follows.

- PC contents becomes 0000H.
- IR contents becomes 00H.
- All interrupts, except TRAP, are disabled by resetting IE flip-flop.
- RST7.5, RST6.5, and RST5.5 interrupts are masked (masking is explained later in the chapter).
- SOD pin output becomes (discussed later).
- RST7.5 flip-flop will be reset to 0 (discussed later).

If the RESET_IN* signal is at logic 0, the 8085 sends out a logic 1 on the RESET_OUT pin at the beginning of the next clock cycle. This is an active high signal. RESET_OUT goes to the logic 0 state only in the next clock cycle after RESET_IN* goes to logic 1 state. This RESET_OUT signal is used to reset other chips in the microcomputer system, such as 8255 PPI, 8251 USART, etc. The timing relationship between RESET_IN* and RESET_OUT is shown in Fig. 18.9.

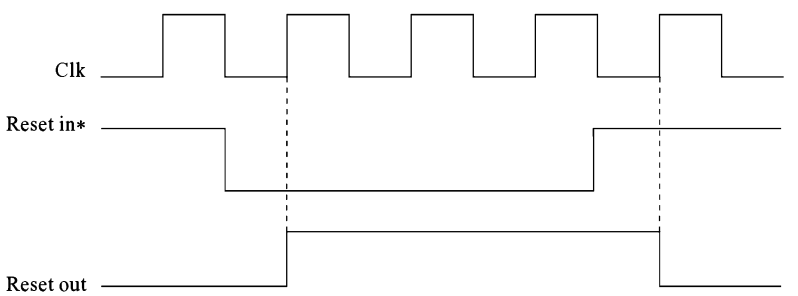


Fig. 18.9
Timing relationship between RESET_IN* and RESET_OUT

Earlier, we have discussed the method of resetting to 0, the IE flip-flop. Now let us discuss the method of setting to 1 the IE flip-flop. It is achieved by the execution of the EI instruction. The EI instruction stands for 'enable interrupts'. It is an 1-byte instruction. When this instruction is executed, the IE flip-flop is set to 1. This enables the 8085 interrupt system. But the interrupts will be enabled only after the next instruction after EI is executed. The reason for this is as follows.

It was stated earlier that when the 8085 branches to an ISS, the interrupt system is disabled automatically. Thus, before we come out of the ISS, it is necessary to again enable the interrupt system using EI instruction. Otherwise, when the control is back with the interrupted program, it cannot be interrupted again, except by TRAP. But immediately after EI, the interrupt system should not get

enabled. If it gets enabled, there is the danger of branching to another ISS before the RET instruction in the current ISS is executed. To overcome this problem, the interrupt system gets enabled only after one instruction is executed subsequent to the execution of the EI instruction. Thus, if we end an ISS with EI followed by RET, the interrupt system is enabled only when the control is back with the interrupted program.

■ 18.4 INTR AND INTA* PINS

INTR is an active high input pin. This is having the lowest priority. It can interrupt the 8085 only if the 8085 is not required to service any other interrupt at the same time. It is a level-sensitive input. The line should remain high till the 8085 checks the internal interrupt signal corresponding to INTR at about the end of an instruction. The INTR internal interrupt signal is activated only when INTR pin is in logic 1 state and IE flip-flop is in logic 1 state, as can be seen from Fig. 18.7. The 8085 gets interrupted because of INTR pin, only if the following conditions are met.

1. INTR internal interrupt signal is active.
2. Higher priority internal interrupt signals are not active at the same time.

If these conditions are not met, even though INTR pin is activated, the 8085 does not get interrupted. In such a case, the INTR line should remain high till the above conditions are met in order to interrupt the 8085.

We always start with a reset of the 8085, because of the switching on of the microcomputer system. Thus, all interrupts except TRAP are disabled to start with. So, if we want INTR pin activation to interrupt the 8085, it is necessary that we have EI instruction in our program. The interrupt system will then be enabled after executing the next instruction after EI. It is important to note that without EI instruction in the main program, the main program will never be interrupted by any of the 8085 interrupts, except TRAP.

INTR is a non-vectored interrupt in 8085. By this we mean that the 8085 does not know by itself the starting address of the ISS. It has to be provided to the 8085 by an external I/O port, or by an interrupt controller like Intel 8259. The action taken by the 8085 when INTR pin is activated is detailed in the following.

18.4.1 Action Taken by 8085 When INTR Pin is Activated

It is assumed that interrupt system is enabled using EI instruction, and higher priority internal interrupt signals are not active.

1. In the penultimate clock cycle of the last machine cycle of every instruction, the 8085 senses all the internal interrupt signals.
2. If INTR internal signal is at logic 1, the 8085 enters an interrupt acknowledge (INA) machine cycle.
3. The interrupt from the I/O port is acknowledged by the 8085 by activating INTA* pin in the T2 state of the INA machine cycle. INTA* is an active low pin. In response to INTA*, the interrupting port should send code for CALL instruction to IR register of 8085 on AD_{7-0} pins. Intel

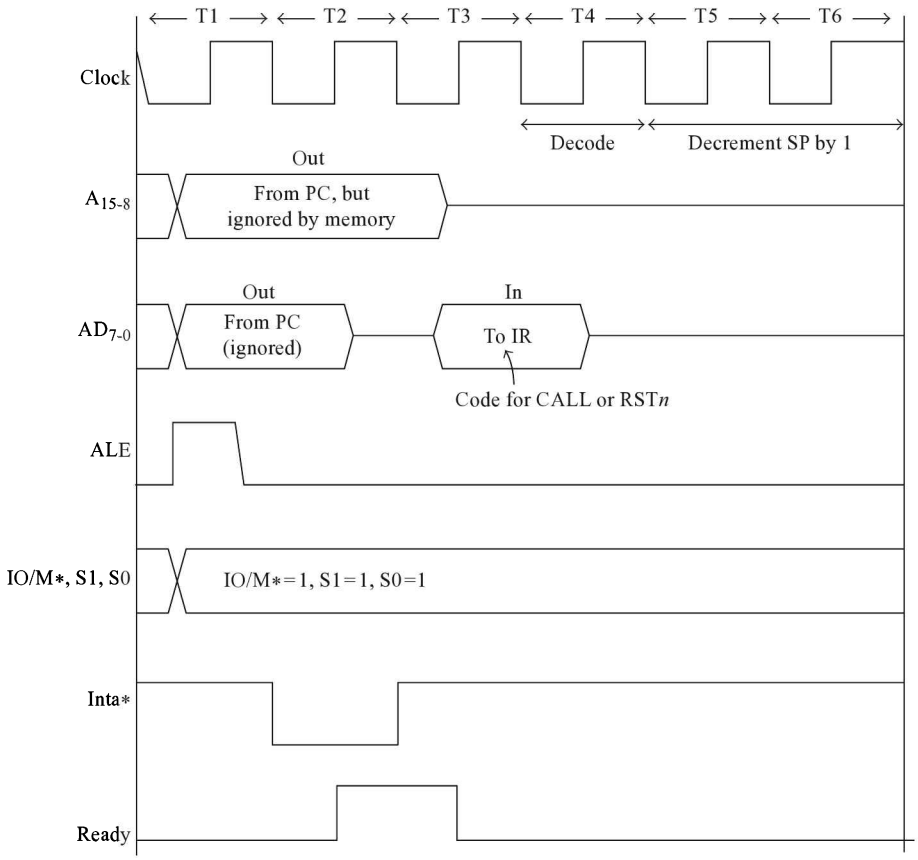


Fig. 18.10 Interrupt acknowledge (INA) machine cycle

8259 interrupt controller is capable of sending such a CALL instruction to 8085. Intel 8259 is discussed in a later chapter. Note that the CALL instruction does not come from memory in this case. So this M1 machine cycle is not an OF machine cycle from memory. Also, it is not an IOR machine cycle, because INTA* is activated, and not RD*. In this INA machine cycle, IO/M*, S1, and S0 will all be in 1 state. This INA machine cycle is shown in Fig. 18.10. INTA* signal becomes a 1 after the 8085 receives the code for CALL during this INA machine cycle. This INA machine cycle that fetches the code for CALL and decodes it, uses six *T* states.

4. In M2 machine cycle, which is also an INA machine cycle, LS byte of ISS address is transferred to the Z register of 8085 by the 8259. This will be in response to activation of INTA* during T2 of this INA machine cycle. This machine cycle uses only three *T* states, as the 8085 does not have to decode this information.
5. In M3 machine cycle, which is also an INA machine cycle, MS byte of ISS address is transferred to the W register of 8085 by the 8259. This will be in response to activation of INTA* during T2 of this INA machine cycle. This machine cycle also uses only three *T* states, as the 8085 does not have to decode this information. Now the 8085 has received the complete ISS

address. After this, save on the stack top the return address available in PC, before branching to the ISS.

6. In M4, which is a MW machine cycle, MS byte of the PC is pushed onto the stack. This takes three *T* states.
7. In M5, which is also a MW machine cycle, LS byte of PC is pushed onto the stack. At this point, the address specified in the CALL instruction, which is present in the WZ register pair, is moved to the PC. All this takes three *T* states. This results in the starting of the ISS execution.

It can be noticed from these points that, after completion of the instruction during which the 8085 got interrupted, it takes 6 + 3 + 3 + 3 + 3 = 18 *T* states before the 8085 transfers control to the ISS.

Incidentally, digressing a bit, it may be appropriate to describe the fetching and execution of a 3-byte CALL instruction stored in memory. This also needs 18 *T* states, but the machine cycles involved are: six *T*-state OF, followed by two MR machine cycles each of three *T* states, and two MW machine cycles each of three *T* states.

18.4.2 Alternative Action by 8085 When INTR Pin is Activated

In response to activation of INTA*, instead of CALL instruction, an RST instruction can be transferred to the IR register of 8085 by an I/O port. The RST instruction is only an 1-byte instruction, and so there will be saving of time in receiving the instruction. The action taken by 8085 during this alternative will be as follows.

It is assumed that the interrupt system is enabled using EI instruction, and higher priority internal interrupt signals are not active.

1. In the penultimate clock cycle of the last machine cycle of every instruction, the 8085 senses all the internal interrupt signals.
2. If INTR internal signal is at logic 1, the 8085 enters an INA machine cycle.
3. The interrupt from the I/O port is acknowledged by the 8085 by activating INTA* pin in the T2 state of INA machine cycle. In response to INTA*, an I/O port should send code for an RST*n* (*n* = 0 to 7) instruction to IR register of 8085 on $AD_{7\text{-}0}$ pins. An I/O port chip like Intel 8255 is capable of sending such an RST*n* instruction to 8085. Intel 8255 is discussed in a later chapter. Note that the RST*n* instruction does not come from the memory in this case. So this M1 machine cycle is not an OF machine cycle from memory. Also, it is not an IOR machine cycle, because INTA* is activated, and not RD*. In this INA machine cycle, IO/M*, S1, and S0 will all be in 1 state. This INA machine cycle has already been shown in Fig. 18.10. INTA* signal becomes a 1 after the 8085 receives the code for RST*n* during this INA machine cycle. This INA machine cycle that fetches the code for RST*n* and decodes it, uses six *T* states. Now the 8085 knows that it has to transfer control to the ISS at location *n* * 8. For example, if the instruction received was RST6, the control should be transferred to the ISS at 6 * 8 = 0030H. This address will be internally stored in the WZ register pair by the control unit of 8085. Now it is required to first of all save on the stack top the return address available in the PC, before branching to the ISS.
4. In M2, which is a MW machine cycle, MS byte of PC is pushed onto the stack. This takes three *T* states.
5. In M3, which is a MW machine cycle, LS byte of PC is pushed onto the stack. At this point, the ISS address, which is present in the WZ register pair, is moved to the PC. This takes three *T* states. This results in the starting of the ISS execution.

It can be noticed from these points that, after completion of the instruction during which the 8085 got interrupted, it takes 6 + 3 + 3 = 12 *T* states before the 8085 transfers control to the ISS.

Generally at location 6 * 8 we will not have the ISS. For example, in the ALS kit there is JMP FFA8H instruction in the 3 bytes starting at 6 * 8 = 0030H. FFA8H is a system RAM location in ALS kit. The system RAM area is used by the kit for storing system data. As such, the ISS cannot start at location FFA8H also. In this system RAM area, 3 bytes starting from FFA8H are reserved for the user to store a jump instruction, say C900H. In such a case, the ISS for INTR when RST6 is received in response to INTA*, starts at location C900H. This is illustrated in Fig. 18.11.

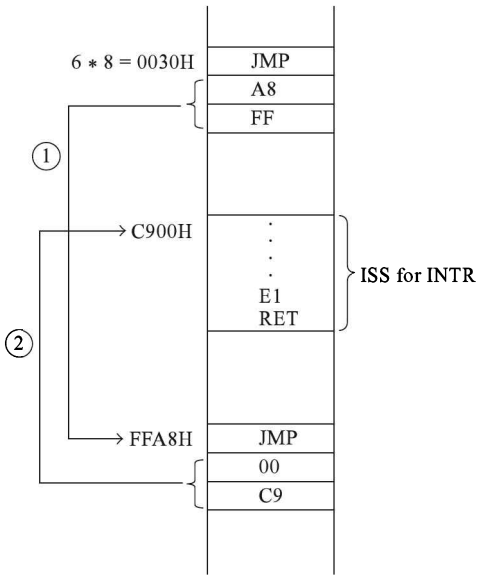


Fig. 18.11
Jumping to ISS when RST6 is received by 8085

Incidentally, digressing a bit, it may be appropriate to describe the fetching and execution of the single-byte RST*n* instruction stored in the memory. This also needs 12 *T* states, but the machine cycles involved are: six *T*-state OF, followed by two MW machine cycles each of three *T* states.

■ 18.5 RST5.5 AND RST6.5 PINS

Both RST5.5 and RST6.5 pins are level-sensitive inputs. RST6.5 has a higher priority than RST5.5. RST5.5 in turn has higher priority than INTR. The function of RST5.5 and RST6.5 are very similar. These pins should remain high till the 8085 checks all the internal interrupt signals at about the end of an instruction. As can be easily seen from Fig. 18.7, the RST5.5 and RST6.5 internal interrupt signals are activated only when

1. These external interrupt pins are in logic 1 state;
2. IE flip-flop is in logic 1 state;
3. These interrupts are not masked by the SIM instruction.

The 8085 gets interrupted because of RST5.5 or RST6.5 pin, only if the following conditions are met.

1. RST5.5 or RST6.5 internal interrupt signal is active;
2. Higher priority internal interrupt signals are not active.

If these conditions are not met, even though RST5.5 or RST6.5 pins are activated, the 8085 does not get interrupted. In such a case, these pins should remain high till these conditions are met in order to interrupt the 8085.

As we always start with a reset of the 8085, because of the switching on of the microcomputer system, all interrupts except TRAP are disabled to start with. So, if we want these pins to interrupt the 8085, it is necessary that we have EI instruction in our program. The interrupt system will then be enabled after executing the next instruction after EI.

RST5.5 and RST6.5 are vectored interrupts in 8085. By this we mean that the 8085 knows by itself the starting address of the ISS. It is 5.5 * 8 = 002CH in the case of RST5.5, and 6.5 * 8 = 0034H in the case of RST6.5. The action taken by 8085 when RST5.5 or RST6.5 is activated, is discussed a little later.

Generally at locations 5.5 * 8 and 6.5 * 8 we will not have the ISS. For example, in the ALS kit there is JMP 05C3H instruction in the 3 bytes starting at 5.5 * 8 = 0024H. 05C3H is an EPROM location in ALS kit. The EPROM contains the monitor program. At 05C3H in the EPROM there is ISS for RST5.5. In the ALS kit, whenever the user presses a key on the keyboard (except RESET key and VECT INTR key), the RST5.5 pin of 8085 is activated. Thus, in ALS kit the RST5.5 interrupt pin is used by the keyboard to interrupt the 8085. Similarly, at location 6.5 * 8 we will not have the ISS. For example, in the ALS kit there is JMP FFABH instruction in the 3 bytes starting at 6.5 * 8 = 0034H. FFABH is a system RAM location in ALS kit. The system RAM area is used by the kit for storing system data. As such, the ISS cannot start at location FFABH also. In this system RAM area, 3 bytes starting from FFABH are reserved for the user to store a jump instruction, say CC00H. In such a case, the ISS for RST6.5 starts at location CC00H.

■ 18.6 RST7.5 PIN

RST7.5 pin is an edge-sensitive input. This is used by peripherals that send a pulse, rather than a sustained high level, for interrupting the processor. Internal to 8085 there is a flip-flop connected to RST7.5 interrupt pin. This flip-flop is set to 1, when a positive-going edge occurs on RST7.5 input. This can be visualized from Fig. 18.7. The waveform for RST7.5 pin and Q output of RST7.5 flip-flop is shown in Fig. 18.12.

RST7.5 internal interrupt signal has a higher priority than the internal interrupt signals of RST6.5, RST5.5, and INTR. As can be seen from Fig. 18.7, the RST7.5 internal interrupt signal is activated only when

1. Q output of the RST7.5 flip-flop is at logic 1;
2. IE flip-flop is in logic 1 state;
3. RST7.5 interrupt is not masked by SIM instruction.

The 8085 gets interrupted because of RST7.5 pin, only if the following conditions are met.

1. RST7.5 internal interrupt signal is active;
2. TRAP internal interrupt signal is not active.

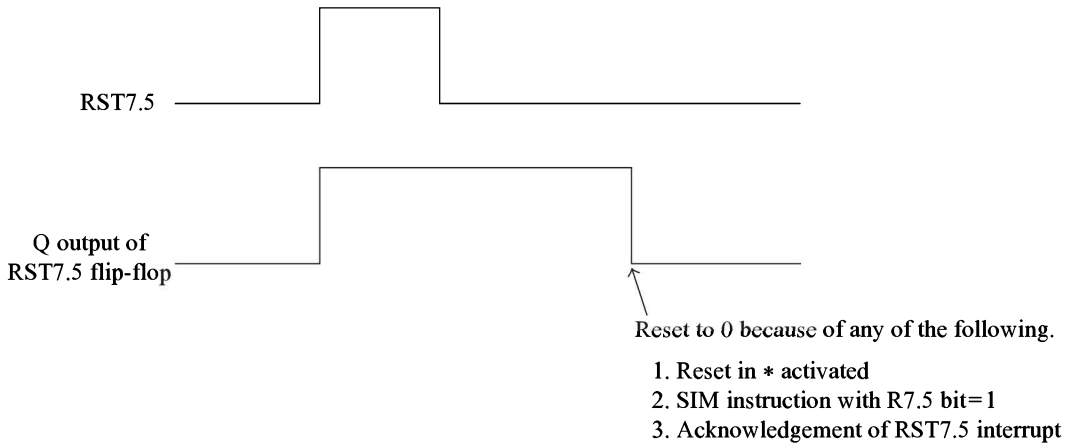


Fig. 18.12 Waveform of RST7.5 pin and RST7.5 flip-flop

If these conditions are not met, even though a rising edge occurs on RST7.5 pin, the 8085 does not get interrupted. As we always start with a reset of the 8085, because of the switching on of the microcomputer system, all interrupts except TRAP are disabled to start with. So, if we want the RST7.5 pin to interrupt the 8085, it is necessary that we have EI instruction in our program. The interrupt system will then be enabled after executing the next instruction after EI.

RST7.5 is a vectored interrupt in 8085. By this we mean that the 8085 knows by itself the starting address of the ISS. It is 7.5 * 8 = 003CH. The action taken by 8085 when RST7.5 is activated is discussed a little later.

Generally at location 7.5 * 8 we will not have the ISS. For example, in the ALS kit there is JMP FFB1H instruction in the 3 bytes starting at 7.5 * 8 = 003CH. FFB1H is a system RAM location in ALS kit. As such, the ISS cannot start at location FFB1H also. In this system RAM area, 3 bytes starting from FFB1H are reserved for the user to store a jump instruction, say CD00H. In such a case, the ISS for RST7.5 starts at location CD00H.

There is no pin of 8085 that can be used to reset the RST7.5 flip-flop. So, RST7.5 flip-flop has to be reset by software. The SIM instruction, to be discussed a little later, can be used to reset the RST7.5 flip-flop. Also, the moment when the 8085 recognizes RST7.5 interrupt request and branches to ISS, the RST7.5 flip-flop is automatically reset to 0.

■ 18.7 TRAP INTERRUPT PIN

Trap is a non-maskable interrupt. This implies that whenever this pin is activated, the 8085 will always get interrupted, even if the 8085 is in DI state. Trap input is both edge- and level-sensitive. Thus the Trap line must make a transition from 0 to 1, and must remain in 1 state till the end of the execution of an instruction in order to interrupt the 8085. Figure 18.13 shows the conditions for which Trap input can cause interrupt of 8085. In this figure t_0 indicates the time at which an instruction execution starts and t_1 indicates the time at which it ends.

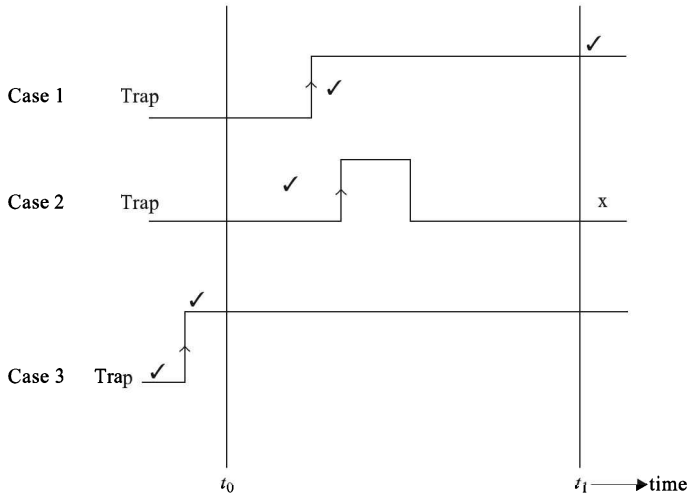


Fig. 18.13
Condition for trap to interrupt the 8085

In Fig. 18.13, only in case 1, the Trap input interrupts the 8085 at time t_1. This is because both the edge and level conditions are satisfied. In case 2, the level condition is not satisfied at time t_1. In case 3, the edge condition is not satisfied at time t_1. However in case 3, 8085 is interrupted because of interrupt at t_0 itself.

At this point it is appropriate to note that an interrupt input is deactivated by an I/O port only after the port is serviced by the 8085. The port needs to perform data transfer with 8085 in the ISS in order that the port is satisfied. This generally happens at about the middle of an ISS. This is in view of the instructions to save the registers in the beginning of the ISS, and instructions to restore registers at the end of the ISS. Thus, an interrupt pin is deactivated by an I/O port only in the middle of the ISS.

Now, it should be clear as to why the Trap input should be both edge- and level-sensitive. If it were to be just edge-sensitive, it would have been susceptible to noise. If it were to be only level-sensitive the problem would be as follows. The Trap pin would still be in the logic 1 state till at about the middle of the ISS, as explained previously. So surely at the end of the first instruction in the ISS, the Trap pin will still be in logic 1 state causing the 8085 to again get interrupted because of the Trap. To solve this problem, Trap is designed to be both edge- and level-sensitive.

Internal to 8085 there is a flip-flop connected to the Trap interrupt pin. As can be seen from Fig. 18.7, the Trap internal interrupt signal is activated whenever a 0 to 1 transition occurs on Trap pin, and the Trap pin remains in the 1 state till the end of an instruction. Trap internal interrupt signal has a higher priority than all the other internal interrupt signals of 8085. So, whenever the Trap internal interrupt signal is activated, the 8085 gets interrupted even if 8085 is in DI state.

The moment when the 8085 recognizes Trap interrupt request and branches to ISS, the Trap flip-flop is automatically reset to 0. It is also reset by activation of reset input of 8085, as can be seen from Fig. 18.7. Thus, even if the external Trap signal remains at logic 1, it cannot further interrupt the 8085 in the ISS for Trap. To again interrupt the 8085, the external Trap signal should be brought to logic 0, then it must go to logic 1, and remain at logic 1 till the end of the instruction to interrupt the 8085.

Trap is a vectored-interrupt in 8085. The 8085 knows by itself the starting address of the ISS as 4.5 * 8 = 0024H. Thus, Trap pin could have been named equivalently as RST4.5 pin. However, Intel prefers to refer to it as Trap. This is probably because, in computer terminology, Trap is a non-maskable interrupt, whereas an interrupt is a maskable one. The action taken by 8085 when Trap is activated is discussed a little later.

Generally at location 4.5 * 8 we will not have the ISS. For example, in the ALS kit there is JMP 0182H instruction in the 3 bytes starting at 4.5 * 8 = 0024H.

In the ALS kit, memory location 0182H is in the EPROM. The EPROM contains the monitor program. At 0182H in the EPROM there is ISS for trap. In the ALS kit, the trap interrupt is used for single-stepping through the program, for debugging purposes.

18.7.1 Action Taken When 8085 is Interrupted Due to a Vector Interrupt

We already know that the vector interrupts in 8085 are trap, RST7.5, RST6.5, and RST5.5. Let us say a vector interrupt pin is active. As an example, let us assume that RST6.5 interrupt pin is active. As was discussed earlier, RST6.5 interrupt pin will interrupt the 8085 only when the following conditions are satisfied.

1. RST6.5 interrupt pin is activated;
2. The 8085 interrupt system is enabled;
3. RST6.5 has not been masked, using SIM instruction;
4. Higher priority internal interrupt signals, trap and RST7.5 are not active.

Similarly, the condition for the other vector interrupts to cause an interrupt to the 8085 has already been discussed. However, the action taken by the 8085 when it gets interrupted because of any vector interrupt, is almost the same. Hence, only the action taken when 8085 is interrupted because of RST6.5 interrupt pin is explained as follows.

The 8085 recognizes the RST6.5 interrupt at about the end of the execution of an instruction. At the end of this instruction, the 8085 enters a (BI) machine cycle. In this BI machine cycle, we can imagine that the IR register is loaded with the code for 'RST6.5' instruction. However, there is no such instruction in reality. As such, we can as well imagine that the IR register is loaded with code for CALL instruction, and the WZ register pair is loaded with 6.5 * 8 = 0034H. This is because, RST6.5 is equivalent to CALL 0034H. This happens the moment when the 8085 recognizes RST6.5 interrupt. The 8085 spends three clock cycles for decoding it. Decoding is purely an internal operation for 8085. During this time, there is no activity on address, data, or control bus. Hence the name 'bus idle' for this machine cycle is justified. This BI machine cycle, which is the first machine cycle M1, needs only three clock cycles (compared with the six clock cycles needed for the first INA machine cycle when INTR causes interrupt to 8085). This is because, the opcode does not have to come from memory or I/O ports. There cannot be any wait states in the BI machine cycle. This is because neither memory, nor I/O port is communicating with 8085 in the BI machine cycle.

The 8085 then stores the contents of the PC, which is the return address, on the stack. M2, which is a MW machine cycle, is used for storing PCH (high byte of PC) on the stack top. It takes three clock cycles.

Similarly, M3, which is also a MW machine cycle, is used for storing PCL (low byte of PC) on the stack top. Then the PC is loaded with 0034H from the WZ register pair. This takes a total of another three clock cycles.

Generally, at location 0034H we will not have the ISS. For example, in the ALS kit there is JMP FFABH instruction in the 3 bytes starting at 0034H. FFABH is a system RAM location in ALS kit. The system RAM area is used by the kit for storing system data. As such, the ISS cannot start at location FFABH also. In this system RAM area, 3 bytes starting from FFABH are reserved for the user to store a jump instruction, say CC00H. In such a case, the ISS for RST6.5 starts at location CC00H.

Thus, when the 8085 gets interrupted because of a vector interrupt, there will be a BI machine cycle followed by two MW machine cycles. The only other occasion when the 8085 enters a BI machine cycle is during the execution of 'DAD rp' instruction.

■ 18.8 EXECUTION OF 'DAD rp' INSTRUCTION

Let us digress a bit at this point to discuss about the execution of the 'DAD rp' instruction. As an example, let us study the execution of the DAD B instruction. If content of BC is 5678H, and content of HL is 12A7H, execution of DAD B should result in HL having 691FH, as shown in the following.

(BC) = 56 78H
(HL) = 12 A7H

(HL) = 69 1FH

In the first machine cycle M1, the opcode for DAD B instruction is fetched from the memory into the IR register of 8085. This is then decoded by the 8085 to interpret it as the opcode for DAD B instruction. This OF machine cycle takes a total of 3 + 1 = 4 clock cycles. Now it is required to add the contents of HL and BC register pairs. Thus the data needed for this instruction execution is available within the 8085. It is similar to ADD B instruction, where the data needed for instruction execution is available within 8085. Generally in such cases the execution portion takes only a fraction of a clock cycle. But in the case of DAD B, as the required operation is addition of 16-bit numbers, it needs much larger time. This is because, in 8085 we have only an 8-bit ALU, with accumulator and temp registers providing the two inputs to the ALU.

In the second machine cycle M2, the following actions take place.

1. Accumulator is temporarily stored in the W register;
2. L register contents are moved to the accumulator;
3. C register contents are moved to the temp register;
4. Addition is performed, and ALU output is moved to the L register.

This machine cycle uses up three clock cycles. It is a BI machine cycle because:

1. No address is sent out by 8085;
2. No data is sent out or received from outside;
3. No external control signals are generated by 8085.

In the third machine cycle M3, the following actions take place.

1. H register contents are moved to the accumulator;
2. B register contents are moved to the temp register;
3. Addition with Cy is performed, and the result stored is in H;
4. Accumulator gets the original value from the W register.

This machine cycle uses up three clock cycles. This is also a BI machine cycle because:

1. No address is sent out by 8085;
2. No data is sent out or received from outside;
3. No external control signals are generated by 8085.

Thus, the DAD B instruction needs a total of ten clock cycles. It consists of OF machine cycle (four clock cycles), followed by two BI machine cycles (each of three clock cycles).

■ 18.9 SIM AND RIM INSTRUCTIONS

It was noted earlier that RST7.5, RST6.5, and RST5.5 interrupt pins can be masked or unmasked. But then what is the need for masking?

18.9.1 Need for Masking

Let us say, the 8085 has been interrupted because of RST6.5 pin and has branched to the ISS for RST6.5. Then, even without DI instruction at the beginning of this ISS, all the interrupts except TRAP are disabled. So, even if RST7.5 pin is activated in the middle of the execution of RST6.5 ISS, the 8085 will not get interrupted because of RST7.5. Actually RST7.5 is a higher priority interrupt, but now it cannot interrupt the lower priority ISS of RST6.5! We can hope to solve this problem by specifically having EI instruction at the beginning of the ISS for RST6.5. Now RST7.5 can definitely interrupt the RST6.5 ISS, but the problem is that even RST5.5 can interrupt the RST6.5 ISS!

This problem can be solved using the concept of *masking of interrupts*. It provides the ability to selectively disable the interrupts. An interrupt pin that is masked cannot interrupt, even if the interrupt pin is activated and interrupts are in general enabled using EI instruction. Similarly, an interrupt pin that is unmasked can interrupt, when the interrupt pin is activated and interrupts are in general enabled using EI instruction.

Thus, all that is needed to prevent RST5.5 from interrupting RST6.5 ISS, while allowing RST7.5 to interrupt the RST6.5 ISS is:

- Enable the interrupt system using EI instruction;
- Mask RST5.5 so that it cannot interrupt the ISS;
- Unmask RST7.5 so that it can interrupt the ISS.

In fact, by proper masking/unmasking, it is now possible to prevent RST7.5 from interrupting RST6.5 ISS, while RST5.5 is permitted to interrupt RST6.5 ISS! This way, we can assign higher priority to RST5.5 compared with RST6.5, and assign lower priority to RST7.5 compared with RST6.5. Masking/unmasking of interrupt pins is done using the SIM instruction in 8085.

18.9.2 SIM Instruction

SIM stands for 'set interrupt mask'. It is a single-byte instruction. Each of the instructions of 8085 that we have come across so far have a single purpose. But the SIM instruction is a multi-purpose instruction. It is used for the following purposes.

1. Masking/unmasking of RST7.5, RST6.5, and RST5.5;
2. Reset to 0 RST7.5 flip-flop;
3. Perform serial output of data.

When SIM instruction is executed, how does the 8085 know which interrupt is to be masked or unmasked? The contents of the accumulator decides the action to be taken. Thus, it is essential to load the accumulator with the desired value before SIM instruction is executed. The meaning of the various bits of the accumulator when SIM is executed is as shown in Fig. 18.14. Note that except bit 5,

7	6	5	4	3	2	1	0	← Bit number
SOD	SOE	X	R7.5	MSE	M7.5	M6.5	M5.5	

Fig. 18.14 Contents of the accumulator before SIM

which is a don't care bit, the other bits of the accumulator decide the effect of executing the SIM instruction.

Masking of interrupts: Only the LS 4 bits of the accumulator are used for masking or unmasking of interrupts.

Bit 3: This is the mask set enable (MSE) bit. If this bit = 0, SIM instruction is not being used for masking or unmasking of interrupts. In such a case, the LS 3 bits of the accumulator are not having any useful information. If MSE bit = 1, the SIM instruction is used for masking or unmasking of interrupts. Then the LS 3 bits provide information about masking or unmasking of interrupts.

Bit 2: This is mask RST7.5 (M7.5) bit. This bit is meaningful only if MSE bit = 1. If MSE = 1 and M7.5 = 0, RST7.5 is unmasked. If MSE = 1 and M7.5 = 1, RST7.5 is masked.

Bit 1: This is M6.5 bit, used for masking/unmasking of RST6.5. It is similar to M7.5 bit.

Bit 0: This is M5.5 bit, used for masking/unmasking of RST5.5. It is similar to M7.5 bit.

It may be noted that only RST7.5, RST6.5, and RST5.5 can be masked or unmasked using SIM instruction. TRAP and INTR cannot be masked or unmasked using SIM. TRAP is not allowed to be masked because it is the highest priority interrupt. INTR does not need the facility of masking because it is the lowest priority interrupt. After reset of 8085 RST7.5, RST6.5, and RST5.5 interrupts will be in masked condition.

Reset RST7.5 flip-flop: Bit 4 (R7.5) of accumulator is used for resetting to 0 RST7.5 flip-flop output, when SIM instruction is executed. If R7.5 = 0, SIM instruction is not being used for resetting of RST7.5 flip-flop. Thus, if R7.5 = 0, there is no change in the RST7.5 flip-flop output. If R7.5 = 1, the RST7.5 flip-flop gets cleared. This can be visualized from Fig. 18.15, which is actually a part of Fig. 18.17.

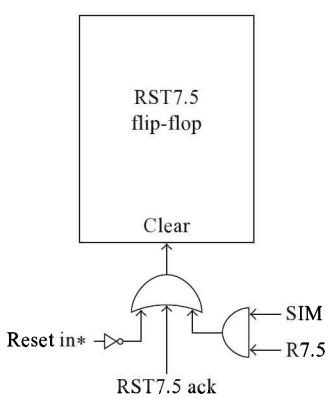


Fig. 18.15
Clearing of RST7.5 flip-flop

In Fig. 18.15, SIM signal is activated when SIM instruction is executed. R7.5 signal is activated when bit 4 (R7.5) of accumulator = 1. So the RST7.5 flip-flop receives logic 1 to its clear input and thus gets cleared when bit 4 of accumulator = 1 and SIM instruction is executed. Alternatively, RST7.5 flip-flop gets cleared when RST7.5 interrupt is recognized by the 8085 or whenever the 8085 is reset.

Serial output of data and SOD pin: Intel 8085 is an 8-bit processor. Thus, it normally sends or receives 8 bits of data at a time. This is known as 8-bit parallel data transfer. In some situations it may not be practical to perform parallel data transfer. Suppose data is to be transmitted over telephone lines, then the 8-bit parallel data will have to be sent out in serial form, one bit after another. Also, we are required to perform serial data transfer when the processor needs to communicate with a serial device, like mouse. Serial communication is much cheaper, because the data transfer takes place on a single line, instead of the eight lines needed in parallel data transfer. However, the speed of communication in serial format is much slower compared with parallel communication. Very few microprocessors provide for on-chip serial communication facility, 8085 being one of them.

Intel 8085 provides SOD (serial output of data) pin and SIM instruction to facilitate serial output of data. From where inside the 8085 does a single bit of information come out on the SOD pin? The MS bit of the accumulator comes out on the SOD pin, when the user commands the 8085 to send out the data. The user commands the 8085 to send out the MS bit of accumulator by making bit 6 of accumulator as 1, and executing the SIM instruction. This can be visualized from Fig. 18.16.

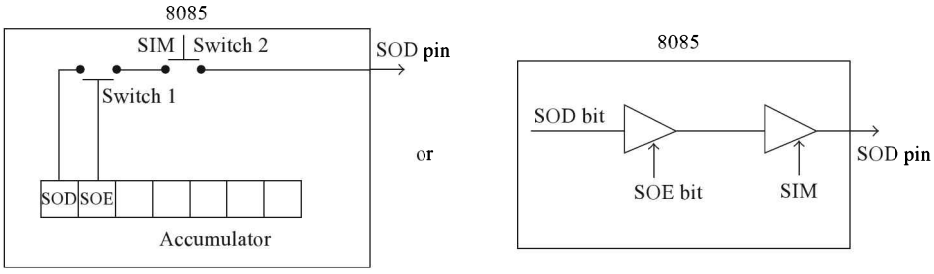


Fig. 18.16 Serial output of data

In Fig. 18.16, switch 1 gets closed when bit 6 of accumulator is at logic 1. Switch 2 gets closed when SIM instruction is executed. So MS bit of the accumulator comes out on SOD pin of 8085 only when bit 6 of accumulator is at logic 1, and SIM instruction is executed. In view of this, the meanings of bits 6 and 7 of the accumulator will be as follows, when we execute SIM instruction.

Bit 6: This is the serial output enable (SOE) bit. If this bit = 0, SIM instruction is not being used for serial output of data. In such a case, the MS bit of accumulator is not having any useful information. If SOE bit = 1, the SIM instruction is used for serial output of data. Then the MS bit provides the data to be sent out on the SOD pin of 8085.

Bit 7: This is serial output data (SOD) bit. This bit is meaningful only if SOE bit = 1. If SOE = 1 and SIM instruction is executed, then the SOD bit comes out on the SOD pin of 8085.

Suppose we have some 8-bit data in the B register, and we want to send it out serially on the SOD pin, starting from the LS bit. Then we have to execute a program that does the following.

1. Bring to MS bit of the accumulator a bit from the B register, starting from the LS bit;
2. Set to 1, bit 6 of the accumulator;
3. Execute the SIM instruction;
4. Repeat these steps eight times in a loop.

18.9.3 RIM Instruction

RIM stands for 'read interrupt mask'. It is a single-byte instruction. Like the SIM instruction, the RIM instruction is also a multi-purpose instruction. It is used for the following purposes.

1. To check whether RST7.5, RST6.5, and RST5.5 are masked or not;
2. To check whether interrupts are enabled or not;
3. To check whether RST7.5, RST6.5, or RST5.5 interrupts are pending or not;
4. To perform serial input of data.

In other words, RIM instruction provides status information about interrupt system and is also used for serial input of data. When RIM instruction is executed, how does the 8085 know which interrupt is masked or unmasked, etc.? The contents of the accumulator after the execution of the RIM instruction provide this information.

Thus, it is essential to look into the accumulator contents after the RIM instruction is executed. The meaning of the various bits of the accumulator after RIM is executed is shown in Fig. 18.17.

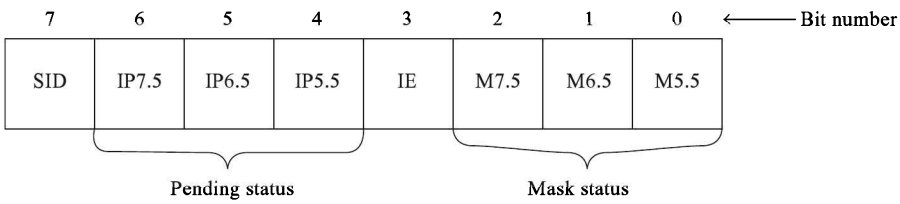


Fig. 18.17 Contents of the accumulator after RIM

Mask status of interrupts: The LS 3 bits of the accumulator are used to provide mask status of interrupts. Note that they are not used for masking or unmasking. Masking or unmasking has to be done using the SIM instruction.

Bit 0: This is mask RST5.5 (M5.5) bit. If this bit = 1, it means that RST5.5 interrupt has been masked. If M5.5 = 0, RST5.5 interrupt is unmasked.

Bit 1: This is mask RST6.5 (M6.5) bit. If this bit = 1, it means that RST6.5 interrupt has been masked. If M6.5 = 0, RST6.5 interrupt is unmasked.

Bit 2: This is mask RST7.5 (M7.5) bit. If this bit = 1, it means that RST7.5 interrupt has been masked. If M7.5 = 0, RST7.5 interrupt is unmasked.

Interrupt enable status: Bit 3 of the accumulator provides the status of IE flip-flop after the RIM instruction is executed. If IE = 1, it means that the interrupt system is enabled. This will be the situation if EI instruction is executed sometime prior to the RIM instruction. If IE = 0, it means that the interrupt system is disabled. This will be the situation if some time prior to execution of the RIM instruction, one of the following things have occurred.

1. DI instruction was executed;
2. Intel 8085 has been reset;
3. Intel 8085 has entered an interrupt service subroutine.

Pending interrupt status: Bits 4, 5, and 6 of the accumulator are used to provide pending interrupt status.

Bit 4: This is interrupt pending RST5.5 (IP5.5) bit. If this bit = 1, it means that RST5.5 interrupt is pending, waiting to be serviced. This will be the situation when RST5.5 interrupt pin is activated, but RST5.5 is masked, or the interrupt system is disabled. If IP5.5 = 0, the RST5.5 interrupt is not pending.

Bit 5: This is interrupt pending RST6.5 (IP6.5) bit. If this bit = 1, it means that RST6.5 interrupt is pending, waiting to be serviced. If IP6.5 = 0, the RST6.5 interrupt is not pending.

Bit 6: This is interrupt pending RST7.5 (IP7.5) bit. If this bit = 1, it means that RST7.5 interrupt is pending, waiting to be serviced. If IP7.5 = 0, the RST7.5 interrupt is not pending.

Serial input of data and SID pin: We have already seen the utility of serial communication. Intel 8085 provides SID (serial input of data) pin and RIM instruction to facilitate serial input of data. The logic state on the SID pin of 8085 enters the MS bit of the accumulator when RIM instruction is executed. This can be visualized from Fig. 18.18.

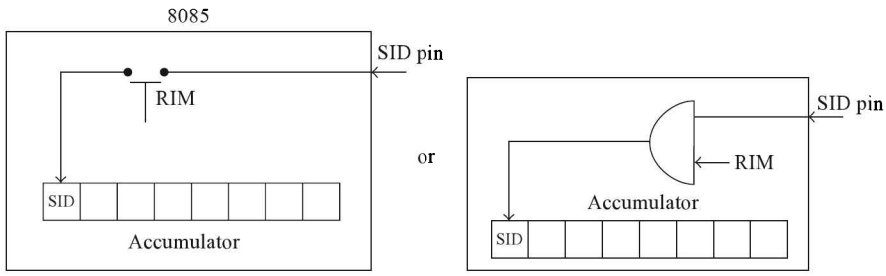


Figure 18.18 Serial input of data

In Fig. 18.18, the switch gets closed when the RIM instruction is executed. So MS bit of the accumulator receives the data present on the SID pin of 8085 when the RIM instruction is executed. In view of this, the meaning of bit 7 of the accumulator will be as follows, after we execute the RIM instruction.

Bit 7: This is SID bit. After the RIM instruction is executed, the data on the SID pin of 8085 gets loaded into this bit position.

Suppose that a serial device is sending some 8-bit data in a serial fashion, starting with the LS bit first. Let us say, we want to receive this information on the SID pin serially and store it in the B register. Then we have to execute a program that does the following.

1. Execute the RIM instruction;
2. Move the MS bit of the accumulator to a bit position in the B register, starting from the LS bit;
3. Repeat these steps eight times in a loop.

Details about the various interrupts of 8085 are summarized in the table that follows.

Pin	*Priority*	*ISS address*	*Sensitivity*	*Condition*
TRAP	Highest	0024H	*1	Unconditional
RST7.5	Second	003CH	Rising edge	Unmasked and EI
RST6.5	Third	0034H	High level	Unmasked and EI
RST5.5	Fourth	002CH	High level	Unmasked and EI
INTR	Lowest	*2	High level	EI

*1: rising edge and high level; *2: supplied by an external device (an I/O port or 8259 PIC).

■ 18.10 HLT INSTRUCTION

HLT is the mnemonic for 'Halt the microprocessor' instruction. It is a 1-byte instruction. When this instruction is executed, the 8085 halts further processing and enters Halt state. This is indicated by S1 and S0 output signals becoming 0 0. The 8085 comes out of the Halt state when a valid interrupt occurs. In such a case, it executes the corresponding interrupt service subroutine and then continues with the instruction after the HLT instruction. However, in most programs the HLT instruction is used for terminating the program. Also activation of reset in * causes 8085 to come out of Halt state.

■ 18.11 PROGRAMS USING INTERRUPTS

To acquaint the reader with the usage of interrupts, a few programs that use interrupts are presented in the following.

18.11.1 Program for Simulation of Throwing a Die

Write an 8085 assembly language program to simulate the throw of a die using an interrupt.

In this program we will have a counter, which counts from 1 to 6, and then repeats the count sequence endlessly in an infinite loop. The throw of the die is simulated by pressing 'Vect Intr' key on the keyboard. On the ALS kit when 'Vect Intr' key is pressed, RST7.5 pin is activated. It interrupts the counter, with the counter having a random value in the range 1 to 6. The 8085 branches to the RST7.5 ISS. In this ISS, the current value of the counter is displayed in the data field, and the control returns to the main program to continue with the counter operation. Thus, every time the 'Vect Intr' key is pressed a random value between 1 to 6 is displayed in the data field, thus simulating the throw of a die.

```
;FILE NAME DIE.ASM

;Main program to reset RST7.5 flipflop, unmask RST7.5, enable interrupts,
;and count from 1 to 6 endlessly in an infinite loop

     ORG C000H
CURDT: EQU FFF9H
UPDDT: EQU 06D3H
DELAY: EQU 04BEH
```

```
        MVI A, 00011011B
        SIM             ;Reset RST7.5 flipflop, Unmask RST7.5
        EI              ;Enable interrupt system

   ;Program segment for an endless counter (1 to 6) loop.
   ;The 2 NOP instructions are needed because interrupt request
   ;lines are sensed by 8085 subsequent to JMP BEGIN instruction
   ;after a short time interval of about 15 clocks. It may be
   ;better to have 3 NOP instructions to provide margin of safety.

BEGIN:MVI A, 01H
LOOP:  NOP
       NOP
       INR A
       CPI 06H
       JNZ LOOP
       JMP BEGIN

   ;RST7.5 ISS to display the counter value

RST75: STA CURDT
       CALL UPDDT ;Display count value in data field
       LXI D, FFFFH
       CALL DELAY ;Generate a delay of 0.5 second
       LDA CURDT
       EI
       RET

   ;When VECT INTR key is pressed, RST7.5 line is activated. So
   ;control is shifted to location 7.5 * 8 = 60 = 003CH. This location
   ;has JMP FFB1H instruction. (For ESA kit there is JMP 8FBFH ). Hence
   ;it is necessary to write JMP RST75 instruction at location FFB1H.
   ;This is done by the following 2 instructions.

       ORG FFB1H; For ESA Kit it should be 'ORG 8FBFH'
       JMP RST75
```

18.11.2 Program for Simulating a Stopwatch

Write an 8085 assembly language program to simulate a stopwatch to display minutes and seconds in the address field. There should be a provision to stop the stopwatch, with the display continuing to show the time just before the stop command.

In this program we will have a counter, which generates a delay of 1 s. After the delay generation, the stopwatch display is incremented by 1 s. This is repeated in a loop. To stop the stopwatch, the user is required to press the 'Vect Intr' key on the ALS kit. On the ALS kit when 'Vect Intr' key is pressed, RST7.5 pin is activated. It interrupts the stopwatch program. The 8085 branches to the RST7.5 ISS. In this ISS, we just have a HLT instruction. Thus, the program comes to a halt, with the display continuing to display the time just before the 'Vect Intr' key is pressed.

```
   ;FILE NAME STOPWACH.ASM
   ;PRESSING THE 'VECT INTR' KEY STOPS THE STOPWATCH, WITH STATIONARY DISPLAY

        ORG C000H
```

```
CURAD:  EQU FFF7H
UPDAD:  EQU 06BCH

RESET:  LXI H,0000H
REPEAT: SHLD CURAD
        CALL UPDAD  ;Display time present in HL in the address field

        MVI A, 00011011B
        SIM           ;Unmask RST7.5, Reset RST7.5 flip flop
        EI            ;Enable interrupt system

        CALL DELAY    ;Generate a delay of 1 second

        LHLD CURAD
        MOV A, L
        ADI 01H
        DAA         ;;;Increment L value in decimal
        CPI 60H
        JZ INC_MIN    ;If L = 60, jump to INC_MIN

        MOV L, A
        JMP REPEAT

INC_MIN:
        MVI L, 00H
        MOV A, H
        ADI 01H
        DAA         ;;;;Make L = 0, and increment H in decimal
        CPI 60H
        JZ RESET       ;If H = 60, jump to RESET

        MOV H, A
        JMP REPEAT

  ;Subroutine to generate a delay of 1 second

  ;To check the proper working of minutes display, load DE with
  ;0444H in this subroutine instead of FFFFH. Then the minutes display
  ;will change every second, so that we can test the proper working in
  ;60 seconds, instead of waiting for 60 minutes.

DELAY:  MVI B, 02H
OUTLOOP:LXI D,FFFFH

INLOOP: DCX D
        MOV A, D
        ORA E
        JNZ INLOOP

        DCR B
        JNZ OUTLOOP
        RET

;RST7.5 Interrupt Service Subroutine

        ORG FFB1H   ;For ESA Kit it should be 'ORG 8FBFH'
        HLT         ;This is the RST7.5 ISS
```

18.11.3 Find the Square of a Number Using a Look-Up Table

Write an 8085 assembly language program to find the square of a single digit (0 to 9) using a look-up table. Display the number and its square in the address field.

The method involves storing a look-up table of squares in RAM, say at location X. Thus, store the following values 00, 01, 04, 09, 16, 25, 36, 49, 64, and 81 in locations starting from X. Then the square of a number is picked up from the look-up table, and displayed in the address field. The program assumes that X address has LS 2 digits as 00H, say C100H.

It waits for the user to press a key in the range 0 to 9. If the user presses any other key, it will remain in the loop. On the ALS kit, whenever the user presses a key other than 'RESET' and 'Vect Intr', RST5.5 interrupt pin is activated.

```
  ;FILE NAME C:\ALS\LUKUPSQR.ASM
       ORG C100H

  X:   DB 00H, 01H, 04H, 09H, 16H, 25H, 36H, 49H, 64H, 81H

       ORG C000H

CURAD: EQU FFF7H
UPDAD: EQU 06BCH
IBUFF: EQU FFFFH

       MVI A, 0EH ;Load A with 0000 1110B
       SIM    ;Unmask RST5.5 i.e. enable keyboard interrupt.

  ;The next 4 instructions check if a key is pressed. If a key is
  ;pressed, RST5.5 pin gets activated but does not interrupt the 8085
  ;as it is in DI state. But RIM instruction execution reveals that
  ;RST5.5 is pending. In such a case, the loop is exited.

AGAIN:
       DI
       RIM
       ANI 00010000B
       JZ AGAIN      ;Stay in this loop till a key is pressed

       EI
       NOP           ;RST5.5 interrupts the 8085 now. Only after NOP is
                     ;executed, interrupt system is enabled.

  ; So control is transferred to RST5.5 ISS. Details of this ISS
  ; is discussed in a later chapter when Intel 8279 chip is discussed.
  ; Execution of this ISS results in location IBUFF getting loaded
  ; with code of key pressed. Then the control is passed on to the
  ; program segment shown below.

       LDA IBUFF
       CPI 0AH
       JNC AGAIN ;;If code is >= 0AH, jump to AGAIN.

       LXI H, X      ;Point HL to the beginning of the look up table.
       MOV L, A      ;Load L from A. Thus, point HL to the location which
                     ;contains the square of the number input by the user.
```

```
        MOV A, M      ;Load A with the square of the number.
        MOV H, L      ;Load H with the number whose square is desired.
        MOV L, A      ;Load L with the square of the number.

        SHLD CURAD
        CALL UPDAD    ;Display the number and its square.
        JMP AGAIN     ;Jump to read the next value from the keyboard.
```

18.11.4 Program for Decimal Down Counter

Write an 8085 assembly language program to implement a decimal down counter (from 99 to 00). The count should be decremented every time a key (other than RESET and VECT INTR key) is pressed. The count is to be displayed in the data field. The counter operation should repeat endlessly in a cyclic manner.

The program works as follows.

1. Load the accumulator with 99.
2. Display the count value present in the accumulator in the data field. RST5.5 is unmasked, and the interrupt system is enabled. The monitor routine RDKBD provided by the manufacturer of the kit is executed. This results in a wait for a key depression, and the moment a key is pressed, its code is loaded in the accumulator, and the control returns to the next instruction after CALL RDKBD. In this program, the key code returned by RDKBD routine is not useful. Hence it is ignored. The count value in the accumulator is now decremented by 1 in decimal notation. This is achieved by adding 99, which is 10's complement of 01. Then control is back with the beginning of this para to count from 99 to 00 cyclically.

```
;FILE NAME C:\ALS\DNCNTR.ASM

        ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
RDKBD: EQU 0634H

        MVI A, 99H ;Initialize A with 99.

REP:    STA CURDT     ;Store A value in CURDT.
        CALL UPDDT    ;Display contents of CURDT in data field.

        MVI A, 00001110B
        SIM           ;Unmask RST5.5
        EI            ;Enable Interrupt system

        CALL RDKBD    ;Wait till a key is pressed and load
                      ;the key code in Accumulator, but ignore it.

        LDA CURDT     ;Reload A from location CURDT.
        ADI 99H
        DAA          ;;Decrement A in decimal notation by
                      ;adding 99, which is 10's complement of 01.
        JMP REP       ;Jump to REP to display the next count.
```

18.11.5 Program for Decimal Down Counter (Alternative)

Write an 8085 assembly language program to implement a decimal down counter (from 99 to 00). The count should be decremented every time VECT INTR key is pressed. The count is to be displayed in the data field. The counter operation should repeat endlessly in a cyclic manner.

```
  ;FILE NAME C:\ALS\DNCNTR1.ASM

       ORG C000H

CURDT: EQU FFF9H
UPDDT: EQU 06D3H
DELAY: EQU 04BEH

       MVI A, 99H    ;Initialize A with 99.

       STA CURDT     ;Store A value in CURDT.
       CALL UPDDT    ;Display contents of CURDT in data field.

       MVI A, 00011011B
       SIM    ;Unmask RST7.5 and reset RST7.5 flip flop
       EI     ;Enable Interrupt system

       ;The next 4 instructions forms an infinite loop during
       ;which VECT INTR key is pressed, causing activation of RST7.5

AGAIN: NOP
       NOP
       NOP
       JMP AGAIN

       ;RST7.5 ISS to decrement count value by 1 in decimal
       ;and display it in data field. Then reset RST7.5 flip flop
       ;and enable interrupt system after a delay of 0.5 seconds.
       ;-------------------------------------------------------

RST75: LDA CURDT
       ADI 99H
       DAA    ;Decrement A register value by 1 in decimal

       STA CURDT
       CALL UPDDT    ;Display count value in data field

       LXI D, 0000H
       CALL DELAY    ;Generate dealy of about 0.5 second
       MVI A, 00010000B
       SIM    ;Reset RST7.5 flip flop
       EI     ;Enable Interrupt system
       RET

       ;-------------------------------------------------------------

       ;The next 2 instructions store JMP RST75 instruction
       ;at FFB1H on the ALS kit

       ORG FFB1H
       JMP RST75
```

18.11.6 Program for Adding 2 Numbers Input from Keyboard

Write an 8085 assembly language program to input two 2-digit hexadecimal numbers from the keyboard, add them and output the result in the address field.

```
;FILE NAME C:\ALS\AD2INPTS.ASM

      ORG C000H
CURAD: EQU FFF7H
UPDAD: EQU 06BCH
CLEAR: EQU 044AH
GTHEX: EQU 052FH

      MVI A, 0EH
      SIM
      EI            ;;;Unmask RST5.5 and enable interrupts

      MVI B, 01
      CALL GTHEX    ;;Input a 2 digit number and display in data field

      MOV A, E
      STA C100H     ;;Store the 2-digit hex number in C100H

      MVI B, 01
      CALL GTHEX    ;;Input a 2 digit number and display in data field

      LDA C100H
      ADD E
      STA CURAD     ;;;Add the two numbers and store in CURAD

      JNC SKIP
      MVI A, 01
      STA CURAD+1   ;;;If Carry =1, store 1 in CURAD+1

SKIP: CALL CLEAR    ;Blank the entire display
      CALL UPDAD    ;Display sum in address field.
      HLT
```

In the above program GTHEX and CLEAR monitor routines were used for the first time. The working of these routines are explained below.

GTHEX monitor routine: It stands for GeT HEXadecimal values from keyboard. This routine is used to get 2 hex digits or 4 hex digits from keyboard. Before calling GTHEX, keyboard interrupt (RST5.5) should be enabled by executing the following instructions.

```
MVI A, 0EH
SIM
EI
```

If contents of B register is 00H, 4 hex digits are received from the keyboard, and they will be displayed in the address field. They are also stored in DE register pair. If more than 4 digits are entered from the keyboard, only the last 4 digits are accepted. For example, if 345678 is entered, it is treated as 5678. If less than 4 digits are entered, like 345 it is treated as 0345. The user entry is terminated with any of the following valid terminators and the keycode of the terminator will be returned in A register.

```
EXEC
NEXT
PREV
```

If contents of B register is 01H, 2 hex digits are received from the keyboard, and they will be displayed in the data field. They are also stored in E register. If more than 2 digits are entered from the keyboard, only the last 2 digits are accepted. For example, if 5678 is entered it is treated as 78. If less than 2 digits are entered, like 5 it is treated as 05. The user entry is terminated with any of the valid terminators as stated above.

On the ALS kit, the GTHEX routine starts at 052FH.

CLEAR monitor routine: This routine blanks out the display in both address and data fields. If contents of B register were 01, a dot will be displayed in the address field. If the contents of register B were 00, even the dot will be blanked out.

All the general purpose registers and Flags register are affected by this routine. On the ALS kit, the CLEAR routine starts at 044AH.

18.11.7 Program for Adding 4 Hex Digits of a 16-bit Number

Write an 8085 assembly language program to add 4 hex digits of a 16-bit number input from the keyboard and display the result in the data field.

```
  ;FILE NAME C:\ALS\AD4INPT.ASM

       ORG C000H
CURDT: EQU FFF9H
UPDDT: EQU 06D3H
GTHEX: EQU 052FH
HXDSP: EQU 05A1H
OBUFF: EQU FFFAH

       MVI A, 0EH
       SIM
       EI            ;;;Unmask RST5.5 and enable interrupts

       MVI B, 00
       CALL GTHEX    ;;Input a 4 digit number and display in address field

       CALL HXDSP    ;Store the 4 hex digits in 4 locations starting from OBUFF

       LXI H, OBUUF
       MOV A, M
       MVI C, 03

AGAIN: INX H
       ADD M
       DCR C
       JNZ AGAIN     ;When we are out of this loop, A will have the sum.

       STA CURDT
       CALL UPDDT    ;Display sum in DATA field.
       HLT
```

In the above program HXDSP monitor routine was used for the first time. The working of this routine is as explained overleaf.

HXDSP monitor routine: This routine converts the value in DE register pair to 4 unpacked hex bytes and stores them in 4 byte locations starting from OBUFF (output buffer) onwards. The LS unpacked hex byte is stored at OBUFF. On ALS kit, OBUFF is location FFFAH, and HXDSP routine STARTS AT 05A1H. Thus, if (DE) = 3456, we will have

FFFAH	06
FFFBH	05
FFFCH	04
FFFDH	03

1. Describe the various methods of parallel data transfer. Bring out their merits and demerits.
2. With a neat diagram describe the architecture of the 8085 interrupt system.
3. Explain EI and DI instructions.
4. Under what circumstances would the 8085 be in the disable interrupt state?
5. Describe the means that are used in a kit to reset the microprocessor.
6. What happens when the 8085 is reset?
7. Describe the action taken by 8085 when INTR is activated.
8. Describe the action taken by 8085 when a vectored interrupt pin is activated.
9. Explain SIM and RIM instructions.
10. Explain the execution of the DAD B instruction.
11. Explain how serial communication can be achieved in a 8085 system.
12. Write a 8085 program, which displays FF in the data field if RST5.5 interrupt is pending, else it should display 00.
13. Write an 8085 assembly language program to implement a decimal down counter (from 99 to 00). The count should be decremented every time VECT INTR key is pressed. The count is to be displayed in the data field. The counter operation should stop after count-down to 00.
14. Write an 8085 assembly language program to implement a decimal down counter (from 99 to 00). The count should be decremented every time a key (other than RESET and VECT INTR key) is pressed. It is assumed that RDKBD monitor routine is not available to the user. The count is to be displayed in the data field. The counter operation should stop after count-down to 00.

19 8212 Non-Programmable 8-Bit I/O Port

- Working of 8212
  - *Pin diagram of 8212*
  - *Intel 8212 in mode 0*
  - *Intel 8212 in mode 1*
- Applications of 8212
  - *Applications of 8212 in mode 0*
  - *Applications of 8212 in mode 1*
- Questions

So far we have discussed in detail about 8085 microprocessor. Interfacing of memory to a microprocessor has also been discussed. Now we will discuss how I/O ports are used in a microcomputer. This chapter will present a detailed explanation of the working and application of the non-programmable I/O port, 8212.

19.1 WORKING OF 8212

I/O ports are of two types. Programmable I/O ports and non-programmable I/O ports. Programmable I/O ports are more popular because their function can be changed by software. There is no need to change the wiring or the hardware to change the function of the I/O port. A very popular programmable I/O port chip is the Intel 8255. It will be discussed in detail in the next chapter.

On the other hand, non-programmable I/O ports require change of wiring or hardware to change its function. An example is Intel 8212, which is the topic of this chapter. As will be seen later, connection has to be changed if the 8212 has to work as an input port instead of an output port. Such non-programmable I/O ports are simple in design.

19.1.1 PIN DIAGRAM OF 8212

The 8212 is available as a dual in-line package chip with 24 pins. Its functional pin diagram is illustrated in Fig. 19.1. The actual pin diagram of 8212 is provided in Fig. 19.2.

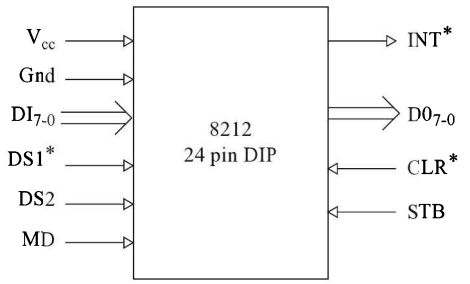


Fig. 19.1
Functional pin diagram of 8212

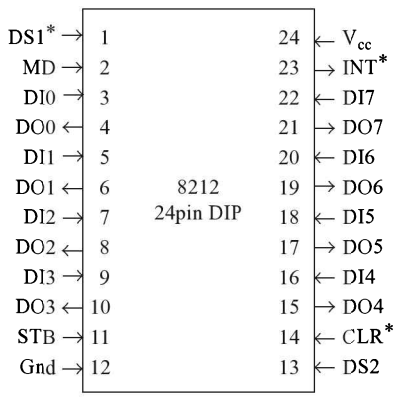


Fig. 19.2
Pin diagram of 8212

It works on a dc supply of +5 V. The 8-bit latch in 8212 receives information present on DI_{7-0} (data inputs). The condition for latching the information present on DI_{7-0} depends on the logic state of the MD (mode) pin. The information present in the latch comes out on DO_{7-0} (data output) pins. The condition to be satisfied for the latched information to come out depends on the MD pin.

There is an edge-triggered D-type flip-flop in the 8212. It is called the service request flip-flop. It is mainly responsible for generation of an interrupt request on INT*. INT* is an active low output pin, which is useful in interrupt-driven data transfer. The internal architecture responsible for activation of INT* signal is shown in Fig. 19.3.

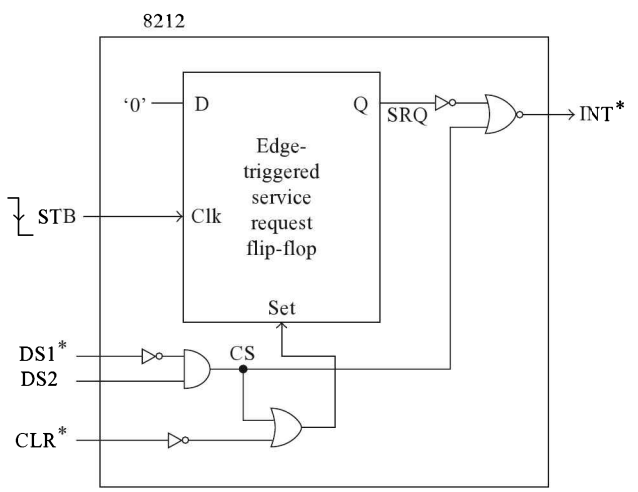


Fig. 19.3
Circuit for activation of INT* in 8212

It can be seen from Fig. 19.3 that INT* is activated in the following two cases.

- When DS1* = 0 and DS2 = 1 so that the internal CS (chip select) signal is activated.
- When STB makes a high to low transition so that the internal SRQ (Q output of service request flip-flop) signal becomes 0.

INT* is deactivated when CS signal = 0 and SRQ = 1. SRQ becomes 1 whenever CLR* = 0 or CS = 1.

Details of connection inside the 8212 for a latch flip-flop are shown in Fig. 19.4. There are eight such latches in 8212. They are all level-triggered D-type flip-flops.

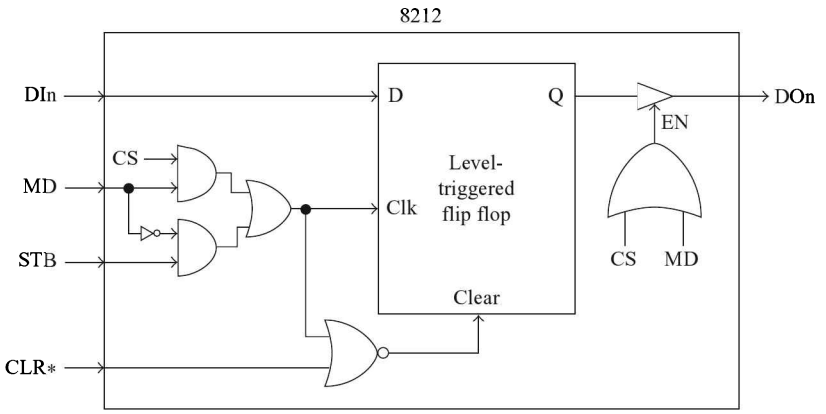


Fig. 19.4 Connection details for a latch flip flop in 8212

The following conclusions can be drawn from Fig. 19.4. If MD = 1, CS acts as the clock for the latches. If MD = 0, STB acts as the clock for the latches. As long as the clock is in a high state, the Q output of the latch follows the corresponding DI input. When the clock makes high to low transition, the data gets latched. The data at the Q output comes out on the corresponding DO pin when the internal EN (enable) signal is activated. The EN signal is activated whenever CS = 1 or MD = 1.

Brief summary of the pins of 8212 is provided in the following.

V_{cc}:	Connected to +5 V dc.
Gnd:	Connected to power supply ground.
MD:	Mode input pin. It determines the source of the clock to the latches. If MD = 0, STB acts as the clock input to the latches. If MD = 1, CS acts as the clock input.
STB:	Strobe input. It is active high input. Used for strobing the data on DI_{7-0} to the latch. When STB makes a high to low transition, INT* becomes 0.
DS1*, DS2:	Device select pins. Can be called chip select pins also. When DS1* = 0 and DS2 = 1, the internal CS signal is activated.
DI_{7-0}:	Data inputs.
DO_{7-0}:	Data outputs. Information present in the latches is sent out by 8212 on these pins when CS = 1 or MD = 1.
INT*:	Interrupt output pin. It is active low output pin. Used for interrupting the microprocessor for performing interrupt-driven data transfer. It is activated whenever STB makes a high to low transition. It is also activated when CS = 1.

CLR*: It is an active low asynchronous clear input. When CLR* is at logic 0 and the clock input to the latch is also at logic 0, it clears the 8-bit latch to 00H. Activation of CLR* deactivates INT* if the device select pins are not active.

19.1.2 Intel 8212 in Mode 0

Figure 19.5 explains the working of 8212 in mode 0.

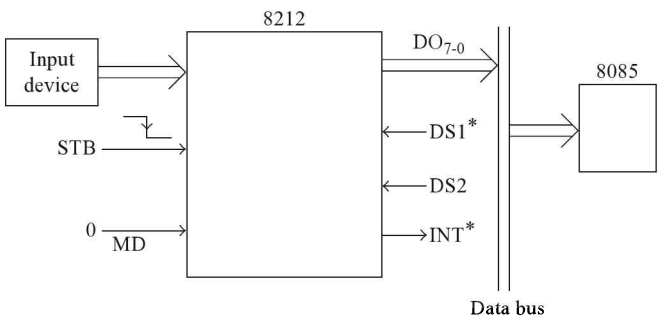


Fig. 19.5
Working of 8212 in mode 0

This mode is generally used when we want the 8212 to function as an input port. An input device may be connected to DI_{7-0}, and the microprocessor can receive the information on DO_{7-0}. In this mode STB acts as the clock for the latches. The 8-bit latch in 8212 follows the information present on DI_{7-0} as long as STB = 1. When STB makes a high to low transition, the 8212 latches the information on DI_{7-0}. High to low transition of STB activates INT* so that interrupt-driven data transfer may be performed if desired. The 8-bit latched information comes out on DO_{7-0} only when the output buffers are enabled. The output buffers are enabled when DS1* = 0 and DS2 = 1.

This mode can also be used when we want the 8212 to function as an output port. In such a case, STB is used by the microprocessor to load the latches with the data to be output. Output device receives the data by activating DS1* and DS2. However, mode 1 operation is more commonly used for output port function.

19.1.3 Intel 8212 in Mode 1

Figure 19.6 explains the working of 8212 in mode 1.

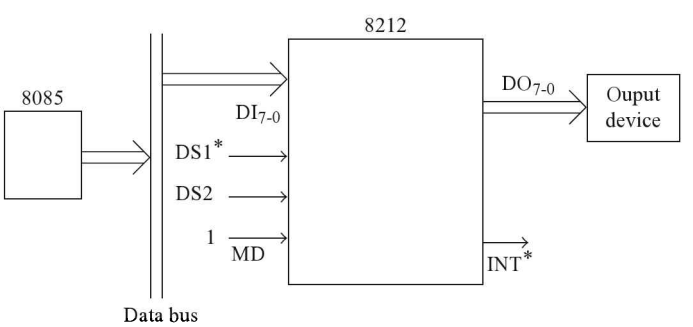


Fig. 19.6
Working of 8212 in mode 1

This mode is generally used when we want the 8212 to function as an output port. In this mode, microprocessor may drive DI_{7-0}, and an output device may receive the information on DO_{7-0}. CS provides the clock to the latches in this mode. The 8212 latches the information on DI_{7-0} when CS makes high to low transition. The activation of CS signal activates INT* so that interrupt-driven data transfer may be performed if desired. The output buffers are always enabled in this mode. So the 8-bit latched information straightaway comes out on DO_{7-0} without any condition.

■ 19.2 APPLICATIONS OF 8212

Intel 8212 can be used in a variety of applications, limited only by the ingenuity of the user. A few applications are indicated below for both mode 0 and mode 1 operations.

19.2.1 APPLICATIONS OF 8212 IN MODE 0

Following applications of 8212 in mode 0 operation will be discussed.

- Gated buffer;
- Bi-directional bus driver;
- Interrupting input port;
- RST*n* interrupt instruction port;
- As supplier of eight RST instructions.

Intel 8212 as gated buffer: Whenever 8212 works in mode 0, it converts a weak logic signal to a strong logic signal. The 8212 outputs in mode 0 are capable of sinking 15 mA in 0 state, and providing a minimum high output voltage of 3.65 V in 1 state. Intel 8212 is driven properly even if the input signal is capable of driving 0.25 mA only. Thus, the 8212 when working in mode 0 acts as a buffer. Intel 8212 as a gated buffer is indicated in Fig. 19.7.

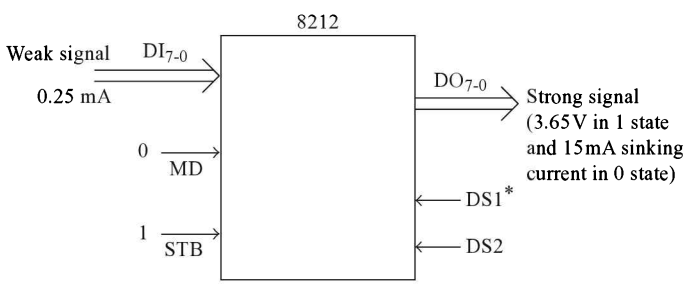


Fig. 19.7
8212 as a gated buffer

We connect STB to logic 1 so that the latch inside 8212 follows the data on DI_{7-0}. It will come out as a strong signal on DO_{7-0} only when DS1* = 0 and DS2 = 1, justifying the name for the application.

Bi-directional bus driver: Sometimes it is necessary to have bi-directional buffering. Such is the case with the data lines of a microprocessor. This can be achieved using two 8212s, as shown in Fig. 19.8.

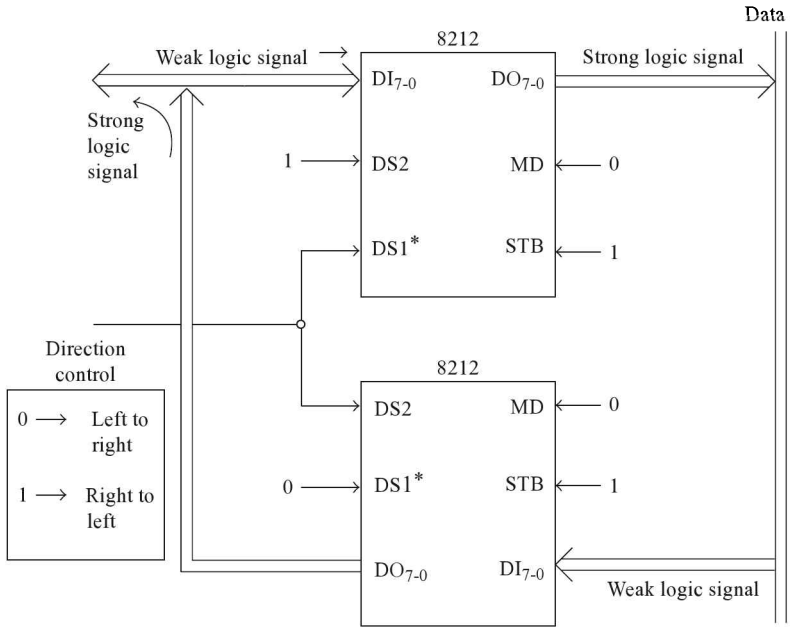


Fig. 19.8
8212 bi-directional bus driver

Both the 8212s are operated in mode 0. It may be noticed that they are connected as gated buffers. When direction control = 0, DS1* becomes 0 for upper 8212, and so the weak logic signal on the left gets transmitted to the right as a strong logic signal.

When direction control = 1, the output buffers of upper 8212 get tristated, but the output buffers of lower 8212 get enabled as DS2 becomes 1. So the weak logic signal from the right gets transmitted to the left as a strong logic signal.

Interrupting input port and RSTn interrupt instruction port: Figure 19.9 illustrates these two applications of 8212. The first 8212 is working as an interrupting input port. It performs data transfer with the microprocessor, after interrupting it. The second 8212 supplies RST*n* code to 8085 in response to INTA* from 8085, when the first 8212 interrupts the 8085 on INTR pin.

When input device has some data on $DI_{7\text{-}0}$, which is to be sent to the accumulator of 8085, it sends a positive pulse on STB of first 8212. When STB makes a high to low transition, first 8212 latches the information on $DI_{7\text{-}0}$. Also the high to low transition of STB causes INT* to become 0. This interrupts 8085 on INTR pin after inversion.

In response to INTR, 8085 sends logic 0 on INTA*. The second 8212 receives it on DS1*. For the second 8212, DS2 is tied to logic 1. On $DI_{7\text{-}0}$ of this 8212, RST*n* code is wired up. As STB of this 8212 is tied to logic 1, RST*n* code is available in the latch of this 8212. Thus when INTA* becomes 0, it activates DS1*, and so the RST*n* code comes out of the second 8212 and enters the IR register of 8085.

Now 8085 branches to $n * 8$. At this location, there is a jump instruction to jump to say, location 2500H. From location 2500H onwards the ISS for INTR is stored. During this ISS execution, IN 50H instruction will be executed. This causes DS1* to become 0 and DS2 to become 1 for the first 8212 because of the chip select circuit shown in Fig. 19.9. Then the latched information in the first 8212 will enter the accumulator of 8085. Once the device selection is deactivated, CS signal makes a high to low transition, which causes deactivation of INT* output of first 8212. This can be explained from Fig. 19.10, which provides timing diagram for INT* output.

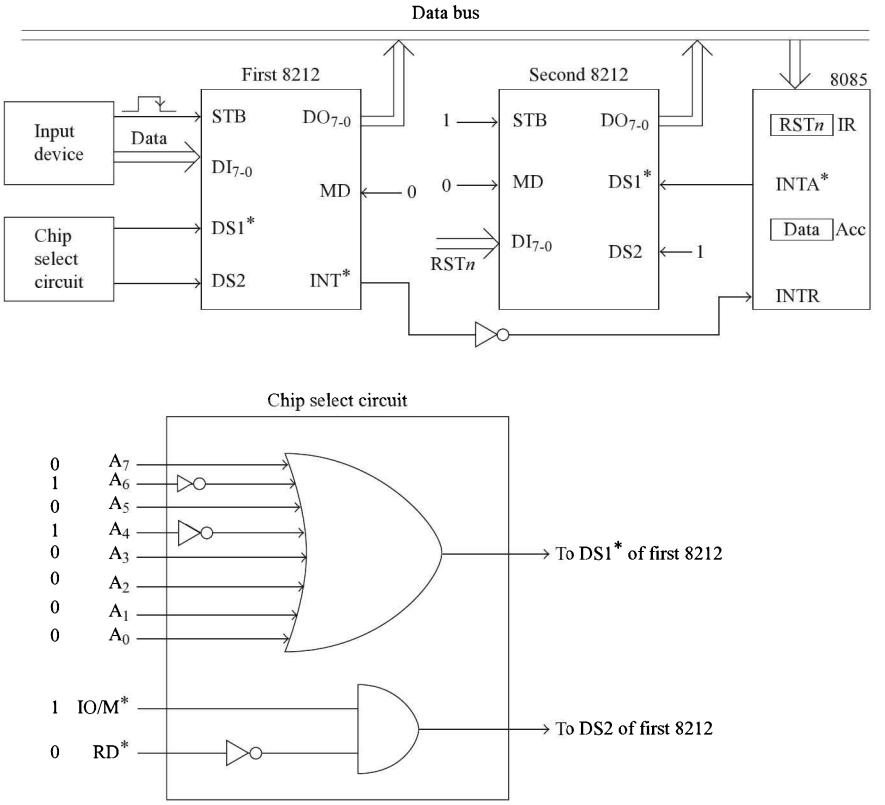


Fig. 19.9 Interrupting input port and RSTn supply port

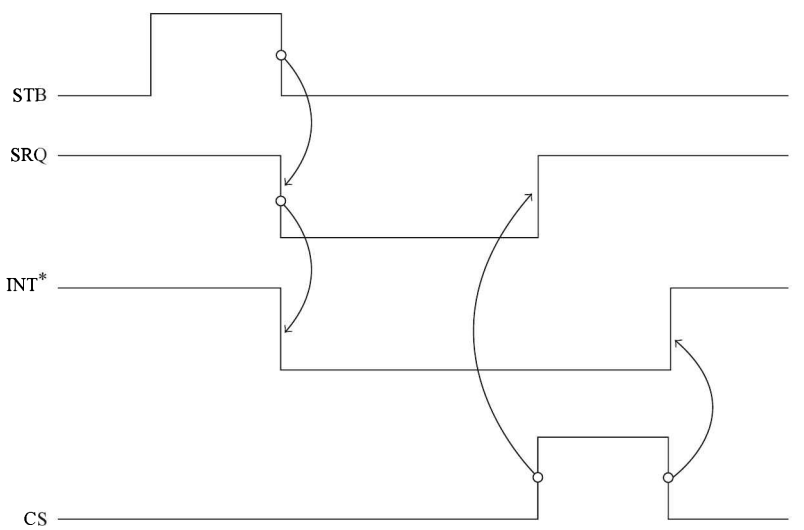


Fig. 19.10 Timing diagram for INT* output

As can be verified from Fig. 19.3, high to low transition of STB causes SRQ output to make a high to low transition. High to low transition of SRQ output causes INT* to be activated. A little later, 0 to 1 transition of CS causes SRQ to go high. Finally, 1 to 0 transition of CS causes INT* to be deactivated.

In the chip select portion of Fig. 19.9, it should be noted that we should connect $A_{7\text{-}0}$ and not $AD_{7\text{-}0}$. Otherwise the chip will get selected wrongly when from some input port the processor reads a data value of 50H. Alternatively, if $AD_{7\text{-}0}$ is not demultiplexed, we can connect $A_{15\text{-}8}$ instead of $A_{7\text{-}0}$, as both will have same address when IO/M* = 1.

As supplier of eight RST instructions: In 8085 we have TRAP, RST7.5, RST6.5, RST5.5, and INTR as the five interrupting pins. Suppose we have more than five I/O devices that would like to perform interrupt-driven data transfer, then on some interrupt pins more than one I/O device will have to interrupt, as shown in Fig. 19.11.

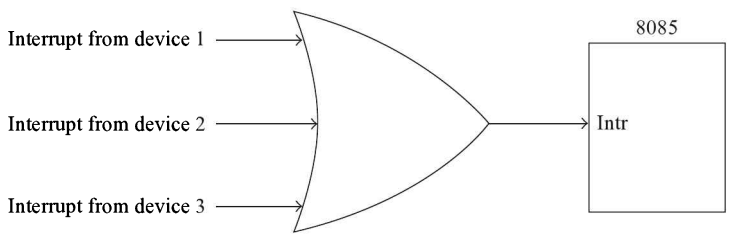


Fig. 19.11 Several devices interrupting on INTR pin

In this case any of the three devices can interrupt the 8085 on INTR pin. So the 8085 does not know the origin of the interrupt request. Thus, in the ISS for INTR, which is now called ILS (interrupt level subroutine), using software the 8085 has to identify the device that needs service. It is essentially checking the interrupt status of each of the devices, till the device that needs service is identified. This process is called 'polling'. Once the device that needs service is identified, branch should take place to the appropriate ISS. However, polling is slow and thus the interrupt response becomes slow.

This problem can be solved to a limited extent in 8085. Upto eight devices are allowed to interrupt on INTR pin and an 8212 supplies automatically a unique RST*n* instruction based on the interrupting device. The interrupting device is identified by hardware in this case resulting in fast interrupt response. This mechanism is illustrated in Fig. 19.12.

Let us say we have eight devices, device 0-7, which would like to perform interrupt-driven data transfer with 8085. Let us say device 0 has the highest priority and device 7 the lowest priority. These devices send their interrupt requests to the clock inputs of eight D-type flip-flops. As can be seen from the figure, whenever a device generates an interrupt request, the Q output of the corresponding D flip-flop is reset to 0. Initially, all these flip-flop outputs are set to 1 state using the ResetOut signal of 8085. These Q outputs are connected to the eight active low inputs of a 74148 priority encoder. In this priority encoder X7 is the highest priority input and X0 is the lowest priority. As such, Q0 output is connected to X7 input. If several inputs are active simultaneously for the priority encoder, the three active low outputs on $A_{2\text{-}0}$ will depend on the highest priority input that is active. Thus, if all the inputs are active, the output on $A_{2\text{-}0}$ will be 000 indicating that X7 is the highest priority input that is active. The relevant truth table for 74148 is provided in Fig. 19.13.

The Q outputs of the D flip-flops are connected to a NAND gate. The output of the NAND gate is connected to INTR pin of 8085. So whenever an interrupt request is generated by an I/O device, the

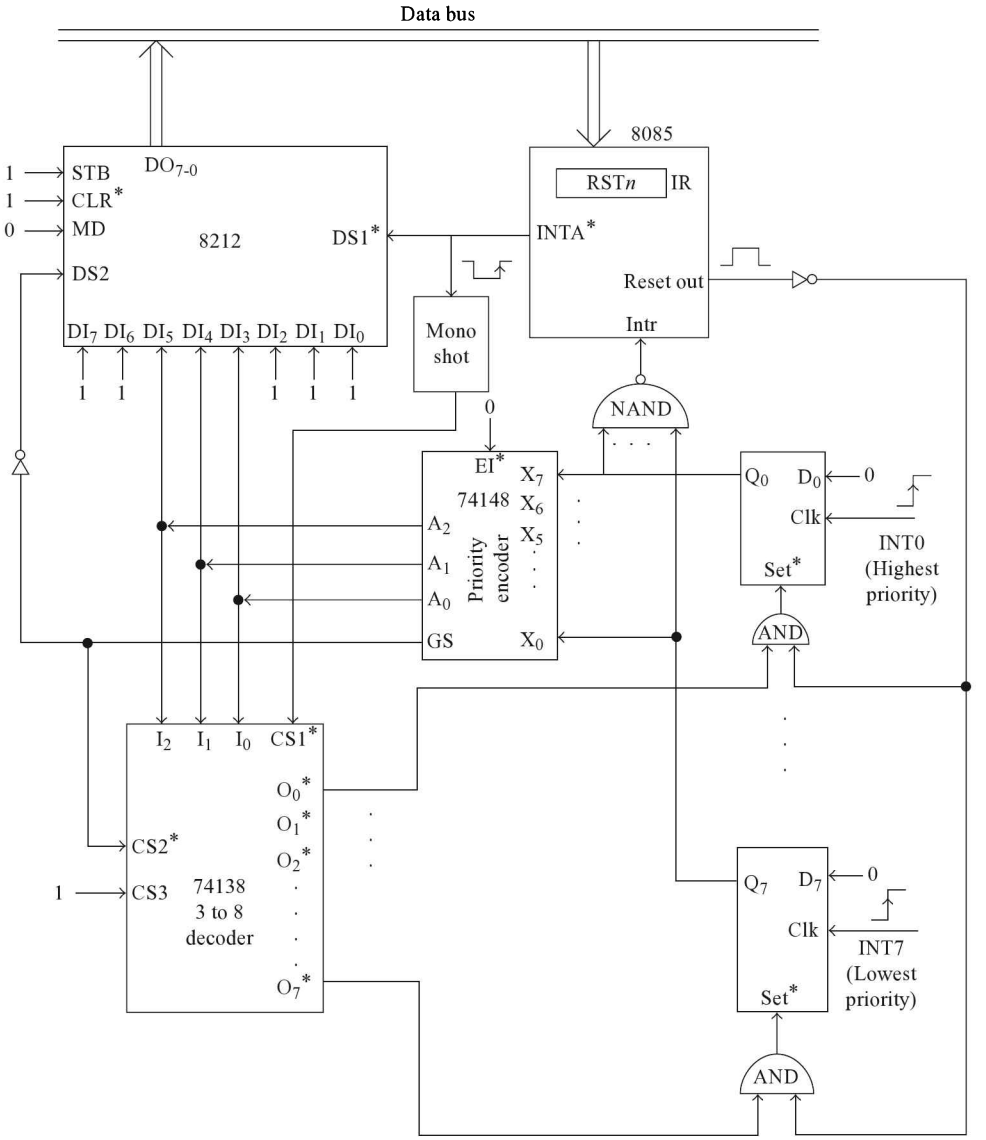
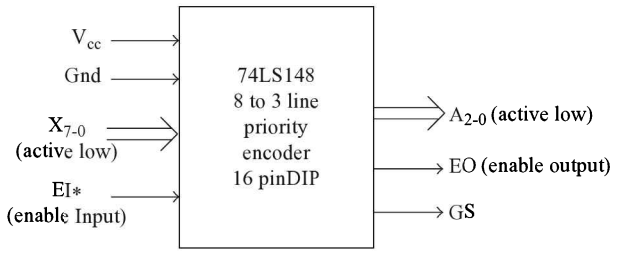


Fig. 19.12 8212 as supplier of eight RST*n* instructions

corresponding Q output is reset to 0 causing the 8085 to be interrupted on INTR pin. The 8085 completes the instruction that is being currently executed, and then generates INTA*.

The INTA* output of 8085 is connected to DS1* input of an 8212. For this 8212, MD pin is connected to logic 0. STB and CLR* are connected to logic 1. DS2 is connected to GS output of 74148 after inversion. Also the data inputs DI_7, DI_6, DI_2, DI_1, and DI_0 are tied to logic 1. DI_5, DI_4, and DI_3 are connected to the A_2, A_1, and A_0 outputs of the priority encoder. The input on $DI_{7\text{-}0}$ corresponds to RST*n* code, as explained in successive paragraphs. So when INTA* is activated, the 8212 sends out on $DO_{7\text{-}0}$, the RST*n* code present on $DI_{7\text{-}0}$. Intel 8085 receives the RST*n* code present on $DO_{7\text{-}0}$ into



X_7 is highest priority input
X_0 is lowest priority input

Active low inputs								Active low outputs		
X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	A_2	A_1	A_0
0	X	X	X	X	X	X	X	0	0	0
1	0	X	X	X	X	X	X	0	0	1
1	1	0	X	X	X	X	X	0	1	0
1	1	1	0	X	X	X	X	0	1	1
1	1	1	1	0	X	X	X	1	0	0
1	1	1	1	1	0	X	X	1	0	1
1	1	1	1	1	1	0	X	1	1	0
1	1	1	1	1	1	1	0	1	1	1

If GS=0, it means A_2, A_1, A_0 outputs are meaningful.
GS=0 if EI*=0 and EO=1.
EO output will be 1 if atleast one of X_7 to X_0 inputs is active, i.e. 0.

Fig. 19.13 Truth table of 74148 priority encoder

the IR register. Now 8085 branches to n* 8. At this location there is a jump instruction to jump to say, location 3500H. From location 3500H onwards the ISS for INTR when device 'n' needs service is stored.

It may be noted that the code for an RST instruction is 11nnn111, where nnn can have values from 000 (for RST0) to 111 (for RST7). Thus, the data inputs of the 8212 have RSTn code depending on priority encoder output. If device 0 interrupt request occurs, the 8212 will have at its data inputs the code for RST 0. If device 4 interrupt request is currently the highest priority interrupt among the interrupt requests that are active, 8212 will receive RST4 code at its data inputs. If only device 7 interrupt request is currently active, 8212 will receive RST7 code at its data inputs.

After device n interrupts the 8085 and the 8085 branches to the appropriate ISS, the corresponding Q output of the D flip-flop should be set to 1, so that INTR becomes deactivated. This is achieved using the 74138 (3 to 8 decoder) chip. It has three active high inputs I_2, I_1, and I_0. These are driven by A_2, A_1, and A_0 outputs of priority encoder. It has eight active low outputs $O_{0\text{-}7}$. If A_2, A_1, A_0 = 000, then O_0 of 74138 will become logic 0, which sets Q_0 to 1. Similarly, if A_2, A_1, A_0 = 111, then O_7 of 74138 will become logic 0, which sets Q_7 to 1. The monoshot in the figure ensures that Q output of D flip-flop is set to 1 only after 8085 has received the RSTn instruction. Connection of GS output of 74148 to CS2* input of 74138 ensures that 74138 is selected only when valid A_2, A_1, A_0 values are output by 74148.

19.2.2 Applications of 8212 in Mode 1

Following applications of 8212 in mode 1 operation will be discussed.

- As low-order address latch;
- As an interrupting output port.

Intel 8212 as low-order address latch: In 8085, low-order address lines and data lines are multiplexed and are available as AD_{7-0}. Many times it is convenient to have address and data separated on different lines. For example, if we want to connect a chip like 2716 (2K×8 EPROM) in a 8085-based system, the 2716 needs 11-bit address on its address pins, and it has separate 8 pins for data. If we connect the AD_{7-0} pins of 8085 to the LS 8 address pins of 2716, the address will be available for only one clock cycle in a machine cycle. Thus, demultiplexing of address and data by latching the low-order address becomes necessary. This can be achieved by using the 8212 as a latch as shown in Fig. 19.14.

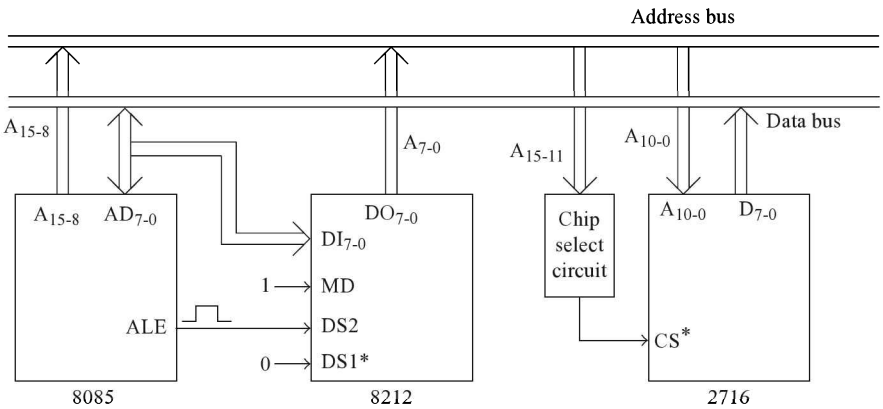


Fig. 19.14 8212 as low-order address latch

In this case, 8212 is used in mode 1. So DO_{7-0} lines are always enabled. DS1* is tied to 0. DS2 is connected to the ALE output of 8085. When address is present on AD_{7-0}, ALE signal is pulsed by 8085 during the first clock cycle T1 of any machine cycle (except BI machine cycle). This causes activation of DS2, and the address information present on DI_{7-0} comes out on DO_{7-0}. After this first clock cycle in the machine cycle ALE goes to 0. This results in latching of address information in 8212. DO_{7-0} is always enabled in mode 1, and so the latched address information is continuously available on DO_{7-0} for the entire machine cycle. This is connected to the address bus.

Only during the second clock cycle T2 of a machine cycle, RD* or WR* signal is activated by 8085. By that time the information on AD_{7-0} will be data, and hence they are connected to the data bus of the microcomputer.

Intel 8212 as an interrupting output port: When an output device desires to perform interrupt-driven data transfer with 8085, it sends a positive going pulse on STB pin of 8212, as shown in Fig. 19.15.

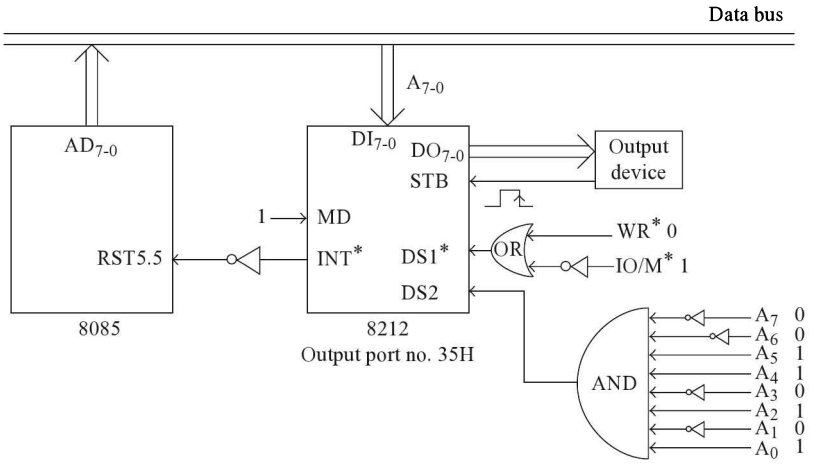
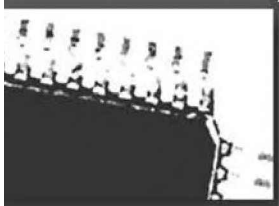


Fig. 19.15 8212 as an interrupting output port

High to low transition on STB results in activation of INT*. This results in 8085 getting interrupted on RST5.5 line as per the figure. Program control is then transferred to location 5.5 * 8. In this location we have a jump instruction to jump to say 2600H. So from 2600H the ISS for RST5.5 starts. During this ISS, OUT 35H instruction will be executed, which momentarily makes DS1* = 0 and DS2 = 1 for the 8212-causing internal CS signal to be pulsed. Then the accumulator contents enter the 8212 and is output to the output device. Also the INT* is deactivated. A little later, execution of the ISS is completed and control returns to the main program.

[redacted]

1. Explain the function of each of the pins of 8212.
2. With a neat diagram explain the working of 8212 in mode 0.
3. With a neat diagram explain the working of 8212 in mode 1.
4. Explain the use of 8212 as gated buffer and bi-directional bus driver.
5. Explain the use of 8212 as an interrupting input port and RST*n* interrupt instruction port.
6. Explain the use of 8212 as a supplier of eight RST instructions.
7. Explain the use of 8212 as a low-order address latch.
8. Explain the use of 8212 as interrupting output port.



20 8255 Programmable Peripheral Interface Chip

- Description of 8255 PPI
  - *Interface with microprocessor*
  - *Interface with I/O devices*
- Operational modes of 8255
- Control port of 8255
  - *Mode definition control word*
  - *Port C bit set/reset control word*
- Mode 1—strobed I/O
  - *Interrupt-driven and status check data transfers*
  - *Interrupt-driven input operation*
  - *Interrupt-driven output operation*
  - *Status check input operation*
  - *Status check output operation*
- Mode 2—bi-directional I/O
  - *Interrupt-driven bi-directional operation*
  - *Status check bi-directional operation*
- Questions

In the previous chapter non-programmable I/O port chip 8212 was discussed. However, programmable peripheral chips are more popular because their setting can be changed by the program, without recourse to change in wiring or hardware. A very popular programmable I/O port chip is the Intel 8255. This will be discussed in detail in this chapter.

20.1 DESCRIPTION OF 8255 PPI

Intel 8255 is a programmable peripheral interface (PPI) chip. It means that it is a programmable chip used for interfacing or connecting peripheral devices. Peripheral device is another name for I/O

device. Also, it is well known that I/O ports are used for connecting I/O devices. Thus, in very simple words, 8255 is a programmable I/O port chip.

The 8255 is available as a 40-pin chip in a dual in line package. Figure 20.1 illustrates the functional pin diagram of 8255. The actual pin diagram of 8255 is provided in Fig. 20.2. Figure 20.3 provides a simplified architecture of 8255. It works on a power supply of +5 V dc. It has two programmable I/O ports each of 8 bits in size, and two programmable I/O ports each of 4 bits in size. They are called Port A, Port B, Port C upper, and Port C lower, respectively. These port pins have the ability to source 1 mA of current at 1.5 V, when programmed to work as output pins. This provides the capability of directly driving Darlington transistors for applications, such as printers and high-voltage displays.

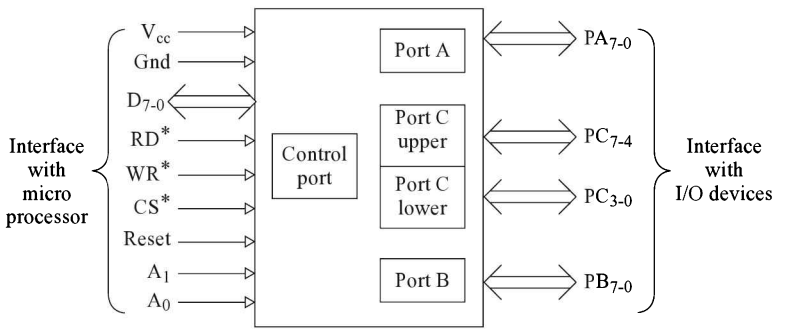


Fig. 20.1
Functional pin diagram of 8255

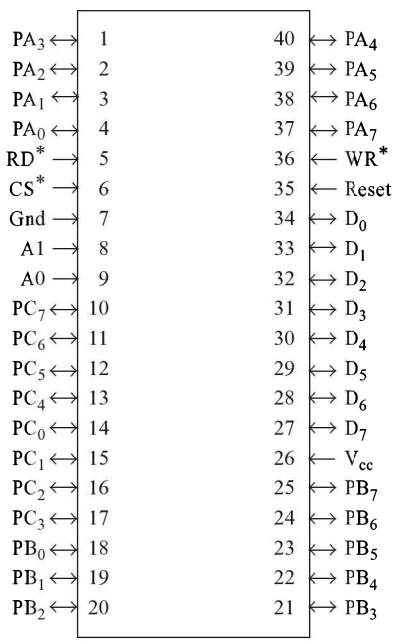


Fig. 20.2
Pin diagram of 8255

Port C upper and Port C lower are addressed as if they constitute a single 8-bit port. This can be verified from Table 20.1, which indicates port selection. Thus Port C can be thought of as being divided into two portions of 4 bits each. They can be independently programmed as input or output lines. So, for example, Port C lower can be programmed as input and Port C upper can be programmed as output. However, in most applications Port C is connected to an 8-bit input device or output device. In such cases both portions of Port C are programmed for the same function.

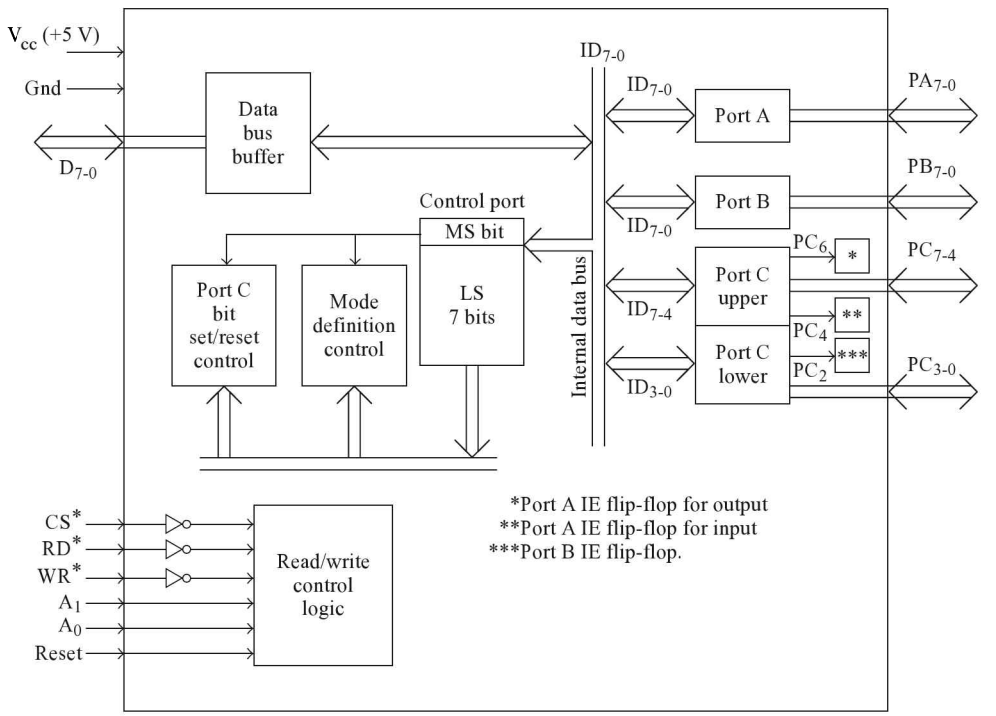


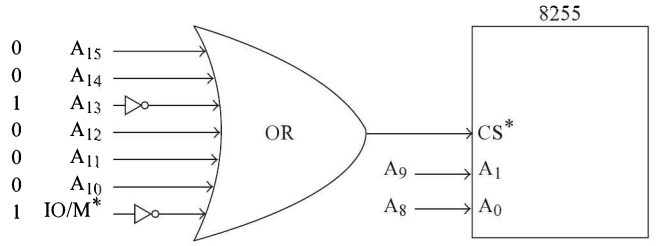
Fig. 20.3 Simplified architecture of 8255

Table 20.1 Port selection in 8255

A_1	A_0	*Port selected*
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control port

Also any line of Port C, which is programmed as output can be set to logic 1, or reset to logic 0 using the single bit set/reset feature of Port C also. This feature reduces software requirement in control-based applications. This facility is provided only for Port C. This feature is also used for enabling/disabling interrupts from 8255 ports, as will be discussed later.

The functionality of these three ports is decided by the contents of the control port. The control port can only be written by the microprocessor. Intel 8085 cannot read it. Thus, there are three ports which can be used for I/O operations, and a control port to control the function of these ports. A port inside the 8255 is selected for communication by the 8085 by the address-input pins A_1 and A_0, as shown in Table 20.1. The direction of data transfer is dictated by the RD* and WR* input signals. Of course, the 8255 chip should be first of all selected by activation of CS* signal before a port inside 8255 can be selected. For example, the control port is written with the contents sent out by 8085 on D_{7-0} pins of 8255 when CS* = 0, WR* = 0, A_1 = 1, and A_0 = 1. Thus, A_1 and A_0 together with RD*, WR*, and CS* decide the manner in which 8085 communicates with 8255.



The address pins in the above circuit could be A_{7-0} instead of A_{15-8}

Fig. 20.4
8255 connected as
I/O-mapped I/O

The 8255 can be connected in a microcomputer system as either memory-mapped I/O or I/O-mapped I/O. Suppose we want 8255 connected as I/O-mapped I/O with addresses of Port A, Port B, Port C, and control port as 20H, 21H, 22H, and 23H, respectively. Then one of the possible chip select circuits is shown in Fig. 20.4. In this figure A_{7-0} could have been used instead of A_{15-8}.

Similarly, suppose we want 8255 connected as memory-mapped I/O with addresses of Port A, Port B, Port C, and control port as FFFCH, FFFDH, FFFEH, and FFFFH respectively. Then one of the possible chip select circuits is shown in Fig. 20.5. In this figure A_{7-0} is the LS byte of address generated using 8212 or 74LS373 as an address latch.

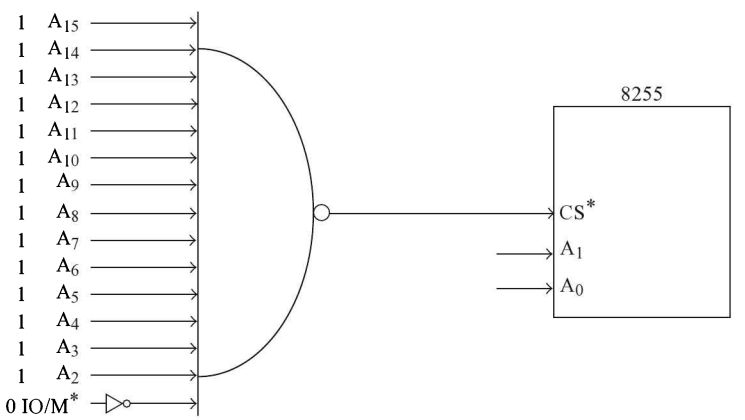


Fig. 20.5
8255 connected as
memory-mapped I/O

20.1.1 Interface with Microprocessor

The pins of 8255 that are used for interfacing with a microprocessor are described in the following:

CS*: It is an active low input pin for 8255. If this pin is at logic 0, the 8255 chip is selected for communication with the microprocessor. If the chip is not selected the data lines D_{7-0} of 8255 will be in tristate.

D_{7-0}: These pins are the data pins, which are used by 8255 for communication with the microprocessor. They are connected to the data bus of the microcomputer system.

RD*: It is an active low input pin for 8255. It is connected to RD* output of 8085. The 8085 activates the RD* input of 8255 when it wants to read the data present in a port of 8255.

WR*: It is an active low input pin for 8255. It is connected to WR* output of 8085. The 8085 activates the WR* input of 8255 when it wants to write data to a port of 8255.

A1, A0: These are address-input pins. They select one of the ports inside 8255 for communication with the microprocessor, as indicated in Table 20.1.

Reset: It is an active high input pin. It is connected to ResetOut output of 8085. It is used to reset the 8255. After a reset of 8255, all the three ports of 8255 work as input ports in mode 0, which is the simplest mode of operation. Operational modes of Ports are described later.

20.1.2 Interface with I/O Devices

The pins of 8255 that are used for interfacing with I/O devices are described in the following.

PA_{7-0}: These eight pins are used by the 8255 for communicating with an I/O device. These pins are output pins if Port A is programmed as an output port. They are input pins if Port A is programmed for input operation.

PB_{7-0}: These eight pins are used by the 8255 for communicating with an I/O device. These pins are output pins if Port B is programmed as an output port. They are input pins if Port B is programmed for input operation.

PC_{7-4}: These four pins are used by the 8255 for communicating with an I/O device. These pins are output pins if Port C upper is programmed as an output port. They are input pins if Port C upper is programmed for input operation.

PC_{3-0}: These four pins are used by the 8255 for communicating with an I/O device. These pins are output pins if Port C lower is programmed as an output port. They are input pins if Port C lower is programmed for input operation.

However, Port C pins can have other functions assigned to them in some cases as will be described next.

■ 20.2 OPERATIONAL MODES OF 8255

Intel 8255 supports three modes of operation. They are mode 0, mode 1, and mode 2.

Mode 0 is called simple I/O or basic I/O, as it is the simplest mode of operation. All of the I/O ports in 8255 are capable of being programmed to work in mode 0. This mode is used with I/O devices whose timing characteristics are clearly known. For example, if we have an input device that wants to send to the microprocessor a byte every 0.5 s, we can execute an IN instruction every 0.5 s to receive the information. In this mode the port inputs are not latched. Thus, the input device must continue with the data on port pins till the port data is read by the microprocessor. So it is useful for reading switch settings, but not useful for reading from a keyboard. Similarly, if we have an output device that wants to receive from the microprocessor a byte every 0.25 s, we can execute an OUT instruction every 0.25 s to send the information. In this mode the port outputs are latched. Thus, the microprocessor is not required to continuously send the data to the port till the output device receives the port data. It is useful, as an example, for sending data to LED display that updates the display based on the latched output.

Mode 1 is called strobed I/O or handshake I/O. This mode is useful when, for example, an input device supplies data to the microprocessor at irregular intervals. In such a case the input device must somehow tell the input port that new data has entered the port. Then the port must inform the processor to read the data. Finally once the processor has read the data, the port must inform the input device that the processor has already read the data. These informations are provided by signals called 'handshake signals'. It is something like the handshake between two strangers, to come to know about each

other before they enter into a dialogue. A port programmed to function in mode 1 uses three handshake signals. Port C provides these handshake signals. Only Port A and Port B can work in mode 1. Port A uses three lines of Port C for handshaking purposes. Port B uses another three lines of Port C for handshaking. Remaining two lines of Port C can be used for simple I/O in mode 0. If only Port A or Port B is working in mode 1, then five lines of Port C are free for use in mode 0.

In mode 0 or mode 1, a port is required to work as an input port or as an output port. It depends on whether an input device or an output device is connected to the port. In contrast with this, mode 2 is called bi-directional handshake I/O. It is useful when the microprocessor sometimes desires to receive information, and at some other times desires to send information to the I/O device connected to 8255. An example is communication with a floppy disk controller card. As mode 2 is bi-directional handshake I/O, it needs more handshake lines than the unidirectional mode 1 strobed I/O. Thus, mode 2 operation makes use of five lines of Port C for handshaking purposes. Only Port A can work in mode 2. The remaining three lines of Port C are used for handshaking if Port B is in mode 1. However, if Port B is functioning in mode 0, these three lines of Port C are used for simple I/O.

In view of this it can be concluded that:

- Port A can work in mode 0, mode 1, or mode 2;
- Port B can work in mode 0 or mode 1;
- Port C works in mode 0 if Port A and Port B are in mode 0. Otherwise, any free lines of Port C, after allocating handshake lines, are used in mode 0.

When the 8255 is reset Port A, Port B, and Port C are initialized to work as input ports in mode 0. This is done to prevent destruction of circuitry connected to a port to which an input device is connected. If ports were initialized as output ports after reset or power-on, a port might try to output to an input device, which might destroy the port, and/or the input device. However, when ports are initialized as input ports, even if output devices are connected to the ports there cannot be any damage to the ports or the output devices.

The contents of the control port decide the 8255 configuration, that is, the way in which the 8255 is programmed to work.

■ 20.3 CONTROL PORT OF 8255

There are two types of command words or control words in 8255. They are:

1. Mode definition control word and
2. Port C bit set/reset control word.

Both these are written to the control port only. From the point of view of the microprocessor there is a single 8-bit control port, which is selected when CS* = 0, WR* = 0, A_1 = 1 and A_0 = 1. But internally there are two control ports, one for mode definition control and another for Port C bit set/reset control. The contents of the control port get latched in mode definition control port if the MS bit of control port = 1. If the MS bit of control port = 0, the contents of the control port gets latched in Port C bit set/reset control port. This can be seen from Fig. 20.3.

20.3.1 Mode Definition Control Word

Figure 20.6 explains the mode definition control word.

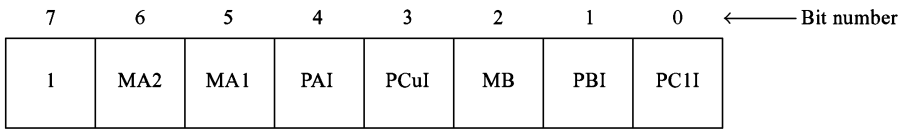


Fig. 20.6 Mode definition control word

Bit 7 must be 1 to indicate that the control port contains mode definition control word. Bit 2 (MB) decides the mode of operation for Port B. Whenever bit 2 = 1, port B works in mode 1 and if bit 2 = 0 port B works in mode 0. Bits 6 and 5 (MA2 and MA1) decide the mode of operation for Port A. Whenever bit 6 = 1, Port A works in mode 2. If bit 6 = 0, bit 5 (MA1) decides whether Port A works in mode 1 or mode 0. As Port C can only work in mode 0, there is no question of configuring the mode for Port C. Bits 0, 1, 3, and 4 decide whether the ports are configured as input or output. For these bits, 1 = Port is configured as input and 0 = Port is configured as output.

Thus, the meaning for the various bits of control port when it contains mode definition control word is as follows.

Bit 0 (PCIl): 1 = Port C lower (PC_{3-0}) as input
0 = Port C lower (PC_{3-0}) as output

Bit 1 (PBI): 1 = Port B as input
0 = Port B as output

Bit 2 (MB): 1 = Port B in mode 1
0 = Port B in mode 0

Bit 3 (PCuI): 1 = Port C upper (PC_{7-4}) as input
0 = Port C upper (PC_{7-4}) as output

Bit 4 (PAI): 1 = Port A as input
0 = Port A as output

Bits 6, 5 (MA2,MA1): 0 0 = Port A in mode 0
0 1 = Port A in mode 1
1 0 = Port A in mode 2
1 1 = Port A in mode 2

Bit 7: Must be 1 to indicate that it is mode definition control.

Example 1: Configure Port A and Port B as input ports, and Port C as an output port, assuming chip select circuit as shown in Fig. 20.4.

It is an I/O-mapped I/O connection with control port address as 23H. All the eight pins of Port C are required to be output pins. This means that Port C is not providing any handshake lines to Port A or Port B. Thus, all the three ports are required to work in mode 0. So the required mode definition control word is as follows.

1 0 0 1 0 0 1 0 = 92H

This control word is written to the control Port by executing the following two instructions.

```
MVI A, 92H
OUT 23H
```

Example 2: Configure Port A as output port, Port B as input port, Port C upper as input port and Port C lower as output port, assuming chip select circuit as shown in Fig. 20.5.

It is a memory-mapped I/O connection with control port address as FFFFH. All the eight pins of Port C are required for data transfer. This means that Port C is not providing any handshake lines to Port A or Port B. Thus, all the three ports are required to work in mode 0. So the required mode definition control word is as follows.

$$1\ 0\ 0\ 0\ 1\ 0\ 1\ 0 = 8AH$$

This control word is written to the control port by executing the following two instructions.

```
MVI A, 8AH
STA FFFFH
```

20.3.2 Port C Bit Set/Reset Control Word

Suppose Port C lower has been configured as output port. If we want to send the data 0101 out on $PC_{3\text{-}0}$ we can execute the following instructions, assuming 8255 is connected as shown in Fig. 20.4.

```
MVI A, xxxx0101B ; x = don't care (it could be 0 or 1)
OUT 22H
```

After a while if we want to send out logic 0 on PC_0 without affecting other lines of Port C, we have to execute the following instructions.

```
IN 22H
ANI 11111110B
OUT 22H
```

Intel 8255 provides an alternative way of sending out logic 1 or 0 on any pin of Port C that is configured as an output pin. It is done using the single bit set/reset feature of Port C. This feature reduces software requirement in control-based applications. This facility is provided only for Port C. Figure 20.7 explains the Port C bit set/reset control word.

7	6	5	4	3	2	1	0	← Bit number
0	x	x	x	SB2	SB1	SB0	S/R*	

Fig. 20.7 Port C bit set/reset control word

Bit 7 must be 0 to indicate that the control port contains Port C bit set/reset control word. Bits 3, 2, and 1 select a bit of Port C that is to be set or reset. Bit 0 decides whether the selected bit of Port C is to be set or reset. Bits 6, 5, and 4 are not used in this control word. They are generally loaded with 000.

Thus, the meaning for the various bits of control port when it contains Port C bit set/reset control word is as follows.

Bit 0 (S/R*): 1 = Set Port C bit selected by bits 3, 2, and 1
0 = Reset Port C bit selected by bits 3, 2, and 1

Bits 3, 2, 1 (SB2, 1, 0): 000 = Select bit 0 of Port C
001 = Select bit 1 of Port C

010 = Select bit 2 of Port C
011 = Select bit 3 of Port C
100 = Select bit 4 of Port C
101 = Select bit 5 of Port C
110 = Select bit 6 of Port C
111 = Select bit 7 of Port C

Bits 6, 5, 4: Are don't cares. Generally loaded with 000

Bit 7: 0 to indicate it is Port C bit set/reset control

Thus, to reset PC_0 using the Port C bit set/reset feature we have to execute only the following two instructions.

```
MVI A, 0 000 000 0B
OUT 23H
```

Similarly, the following two instructions set PC_2.

```
MVI A, 0 000 010 1B
OUT 23H
```

The main use of Port C bit set/reset control word is for enabling or disabling Port A and Port B interrupts when they work in strobed mode operation. This will be discussed later.

■ 20.4 MODE 1—STROBED I/O

Mode 1 is called strobed I/O or handshake I/O. This mode is useful when an input device supplies data to the microprocessor at irregular intervals or an output device desires data from the microprocessor at irregular intervals. A port programmed to function in mode 1 uses three handshake signals. Port C provides these handshake signals. Only Port A and Port B can work in mode 1.

PC_2, PC_1, and PC_0 pins provide the handshake signals for Port B when it is configured as handshake input port or output port. Similarly, PC_5, PC_4, and PC_3 pins provide the handshake signals for Port A when it is configured as handshake input port. However, PC_7, PC_6, and PC_3 pins provide the handshake signals for Port A when it is configured as handshake output port. Note that PC_3 pin will be a handshake line for Port A in input and also output operations. If both Port A and Port B work in mode 1, the remaining two pins of Port C can be used for simple I/O in mode 0. If only Port A or Port B is working in mode 1, then five pins of Port C are free for use in mode 0.

Example 1: Configure Port A as strobed input port, Port B as strobed output port, and PC_7, PC_6 as output lines.

The required mode definition control word is shown in Fig. 20.8.

7	6	5	4	3	2	1	0	← Bit number
1	0	1	1	0	1	0	X	

Fig. 20.8 Mode definition control word for Ex. 1

It is to be noted that the LS bit is a don't care. This is because all the four lines of Port C lower are used for handshaking purposes some of which will be input pins and some others outputs. The 8255 will automatically configure these handshake pins, and so it is unimportant whether this LS bit is a 0 or a 1.

Bit 3 must be a 0 to indicate that PC_7 and PC_6 are output lines. In this case it does not mean that the entire Port C upper is output, as PC_5 and PC_4 are used for handshaking purposes.

The instructions to achieve this requirement assuming the chip select circuit of Fig. 20.4 are as follows.

```
MVI A, B4H; Treating X as 0
OUT 23H
```

Example 2: Configure Port A as strobed output port, Port B as strobed input port, and PC_5, PC_4 as input lines.

The required mode definition control word is shown in Fig. 20.9.

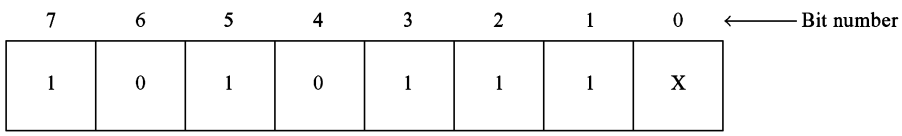


Fig. 20.9 Mode definition control word for Ex. 2

Note that bit 3 must be an 1 to indicate that PC_5 and PC_4 are input lines. In this case it does not mean that the entire Port C upper is input, as PC_7 and PC_6 are used for handshaking purposes.

The instructions to achieve this requirement assuming the chip select circuit of Fig. 20.4 are as follows.

```
MVI A, AFH; Treating X as 1
OUT 23H
```

20.4.1 Interrupt-Driven and Status Check Data Transfers

Among the three pins provided by Port C to a port for the purpose of handshake data transfer, one of them called INT is used for interrupting the microprocessor. Actually Port C provides INT_A as interrupt from Port A and INT_B as interrupt from Port B. These interrupt request outputs of 8255 can be inhibited by resetting to 0 the associated EI flip-flop or enabled by setting to 1 the associated EI flip-flop. This is accomplished by Port C bit set/reset control function. This function allows the user to allow or disallow a specific peripheral to interrupt the 8085, without altering the general interrupt structure.

For example, let us say INT_B is connected to interrupt the 8085 on TRAP pin. Then whenever the TRAP pin is activated, the 8085 is always interrupted. But at some point in the program we may not want the 8085 to be interrupted from Port B. At that point even if DI instruction is executed, it cannot disable the TRAP interrupt. Other interrupts of 8085 also get disabled. But our interest may be to disable only TRAP. To solve this problem, facility is provided in 8255 to disable a Port interrupt, for example, from Port B. Then the INT_B line is never activated, and so the TRAP pin is never activated.

As another example, let us say INT_A is connected to interrupt the 8085 on INTR pin. Then whenever the INTR pin is activated, the 8085 is interrupted. But at some point in the program we may not want the 8085 to be interrupted from Port A. At that point if the DI instruction is executed, it will disable all the interrupts except TRAP. But our interest may be to disable only INTR. To solve this

problem, facility is provided in 8255 to disable a Port interrupt, for example, from Port A. Then the INT_A line is never activated, and so the INTR pin is never activated.

If Port interrupt is enabled, then the data transfer can be performed in the interrupt-driven mode. If Port interrupt is disabled, we have to resort to status check data transfer.

When Port B is programmed to work in mode 1 (input or output), if PC_2 bit is set to 1, Port B EI flip-flop will be set to 1 thus enabling Port B interrupt. This can be seen from Fig. 20.3. It is to be noted that when Port B works in mode 1, PC_2 pin will receive an input handshake signal. The design is such that the logic value on this pin does not enter PC_2 bit. So the PC_2 bit can be set or reset using Port C bit set/reset control without regard to the logic value on PC_2 pin.

If Port B is working in mode 1 input and PC_2 is set to 1, INT_B will be activated when Port B is filled with new data by the peripheral. If PC_2 bit is reset to 0, even if new data is supplied by the peripheral, INT_B will not be activated.

If Port B is working in mode 1 output and PC_2 bit is set to 1, INT_B will be activated when the peripheral empties data from Port B. If PC_2 bit is reset to 0, then even if the peripheral empties data from Port B, INT_B will not be activated.

Similarly, when Port A is programmed to work in mode 1 input, if PC_4 bit is set to 1, Port A interrupt is enabled. It is disabled if PC_4 bit is reset to 0. Finally, when Port A is programmed to work in mode 1 output, if PC_6 bit is set to 1, Port A interrupt is enabled. It is disabled if PC_6 bit is reset to 0. PC_4 and PC_6 bits can be set or reset using Port C bit set/reset control without regard to the logic value on PC_4 and PC_6 pins when Port A is working in mode 1 input and mode 1 output, respectively.

It is to be noted that after a reset of 8255, Ports A and B interrupts are disabled. They are also disabled following the execution of mode definition control word.

20.4.2 Interrupt-Driven Input Operation

Let us say we have an input device that supplies data at irregular intervals. Then we connect it to say Port B of 8255, which is configured to work in mode 1. Figure 20.10 depicts the details. Figure 20.11 provides the timing diagram for this data transfer.

It makes use of three handshake signals supplied by Port C. STB* is an input pin to 8255, and IBF and INT are output pins of 8255 as described in the following.

STB*: It is an active low STRobe input pulse to the 8255. The peripheral sends a low-going pulse with a minimum width of 500 ns on this input of 8255 whenever it has data to send to the port. When STB makes logic 0 to logic 1 transition the peripheral data on the port pins are latched by the port buffer. The data on the port pins should be held for atleast 180 ns after this transition. For Port A, the signal is called STB_A*, and for Port B it is called STB_B*. This handshake pin is the same as PC_4 for Port A and PC_2 for Port B.

IBF: It is an active high output pin of 8255. It indicates input buffer full status to the peripheral and the microprocessor. The buffer is called an input buffer, as it is a strobed input operation. IBF goes to logic 1 within 300 ns after STB* goes to logic 0. IBF goes to logic 0 within 300 ns after the port data is read by the microprocessor. For Port A, the signal is called IBF_A, and for Port B it is called IBF_B. This handshake pin is the same as PC_5 for Port A and PC_1 for Port B.

INT: It is an active high output pin of 8255. This signal is used to interrupt the microprocessor if interrupt-driven data transfer is desired. If status check data transfer is contemplated, this pin is left open without connecting to an interrupt pin of the microprocessor. INT goes

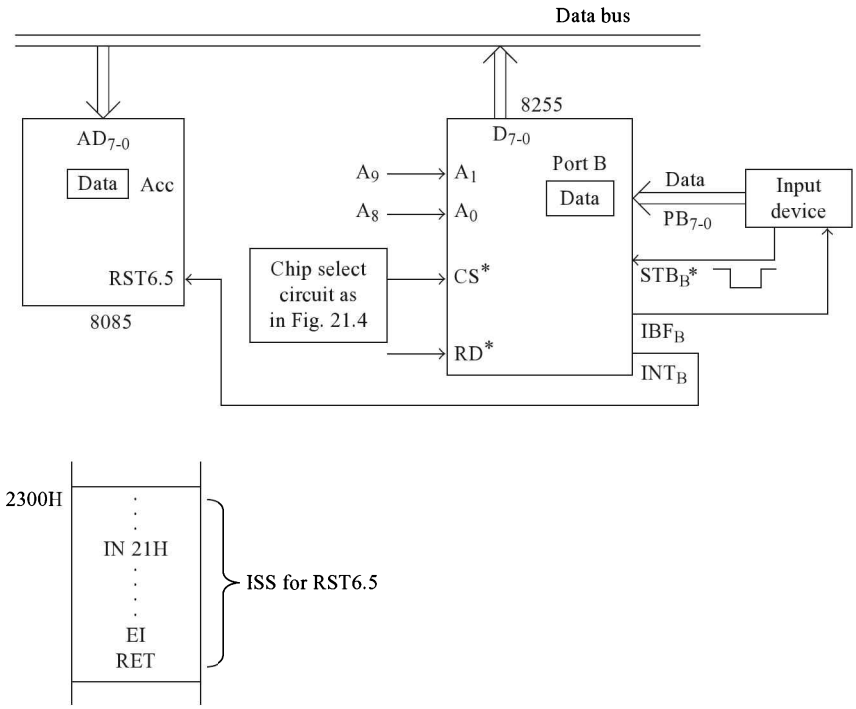


Fig. 20.10 Mode 1 input operation of 8255

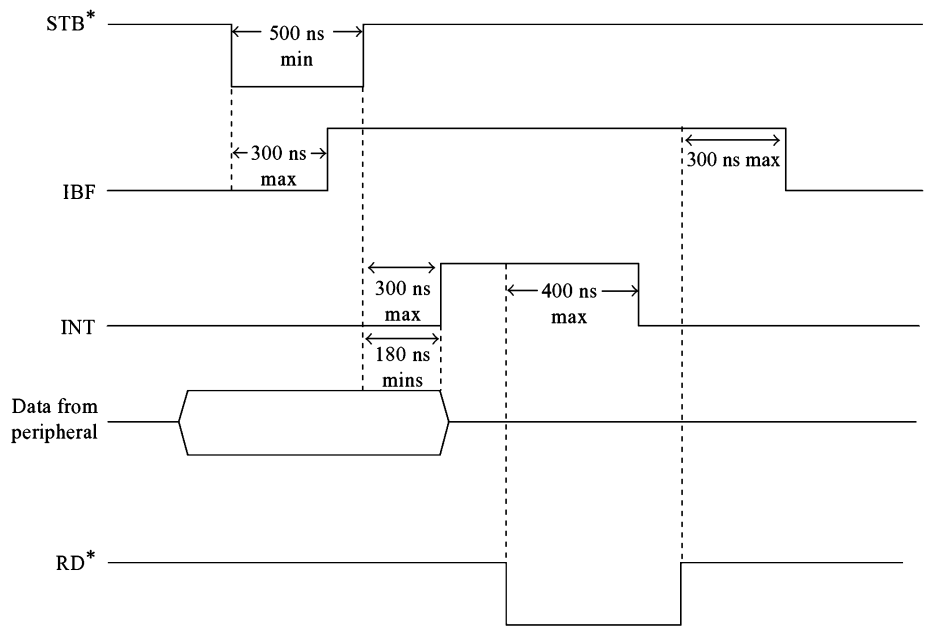


Fig. 20.11 Timing diagram for mode 1 input operation

to logic 1 within 300 ns after STB* goes to logic 1. This is so provided that the IBF signal is active and port interrupt is enabled by setting to 1 the EI flip-flop associated with the port. After a reset of 8255, Ports A and B interrupts are disabled. They are also disabled following the execution of mode definition control word.

INT goes to logic 0 within 400 ns after the RD* input signal goes to logic 0 for reading the port data by the microprocessor. This is true if the Port interrupt was enabled. If Port interrupt is disabled, INT output will always be in logic 0.

For Port A, the interrupt signal is called INT_A, and for Port B it is called INT_B. This handshake pin is the same as PC_3 for Port A and PC_0 for Port B.

From the previous information, it should be clear that PC_2, PC_1, and PC_0 are the handshake signals for Port B when it is configured as handshake input port. Similarly, PC_5, PC_4, and PC_3 are the handshake signals for Port A when it is configured as handshake input port.

For the interrupt-driven input operation, the Port interrupt should be enabled. If we want Port B in mode 1 input (without bothering about other ports), we have to use mode definition control as shown in the following. The chip select circuit for 8255 is assumed to be as in Fig. 20.4.

```
MVI A, 1XXXX11XB; X's are don't care bits
OUT 23H
```

Then, to enable Port B interrupt, PC_2 bit is required to be set to 1 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 010 1B; X's are don't care bits
OUT 23H
```

The interrupt-driven handshake data transfer takes place as follows. Whenever the peripheral has data to be sent to the port, it senses the IBF output of 8255. If IBF = 0, indicating that the input buffer is empty, it sends a low-going pulse on STB*. When STB* makes a 0 to 1 transition, the data on the port pins are latched by the input buffer. By this time the IBF output is activated indicating that the input buffer has become full. Also INT output becomes logic 1. In Fig. 20.10, the 8085 is interrupted on RST6.5 pin because of the INT output. It is essentially indicating to the 8085 that there is data for it to be received from 8255.

Intel 8085 finishes execution of the current instruction. Then it stores the contents of the PC on the stack top, and then it branches to memory location $6.5 \times 8 = 52 =$ 0034H. This of course is true if interrupts are enabled, RST6.5 is unmasked, and finally RST7.5 and TRAP, which are the higher priority interrupts, are not active. At location 0034H there will be a jump instruction to say 2300H, where the actual ISS starts. In the ISS, 8085 executes IN 21H, assuming Port B address to be 21H. This results in activation of CS* and RD* inputs of 8255. Also A_1 becomes 0 and A_0 becomes 1 thus selecting Port B. Thus, the Port B contents are read by the 8085 in the ISS. Since the interrupting port is serviced, INT output of 8255 is deactivated. As the input buffer has been emptied by the 8085, the IBF output is also deactivated. This completes the handshake data transfer by the input device in the ISS.

20.4.3 Interrupt-Driven Output Operation

Let us say we have an output device that needs data at irregular intervals. Then we connect it to say Port B of 8255, which is configured to work in mode 1. Figure 20.12 depicts the details. Figure 20.13 provides the timing diagram for this data transfer.

It makes use of three handshake signals supplied by Port C. ACK* is an input pin to 8255, and OBF* and INT are output pins of 8255 in the description that follows.

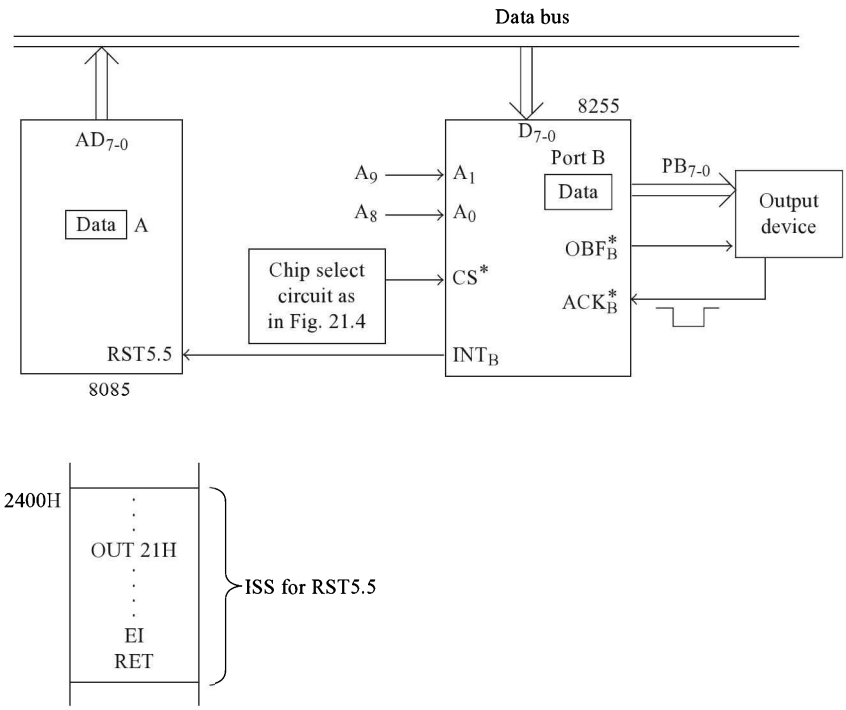


Fig. 20.12 Mode 1 output operation of 8255

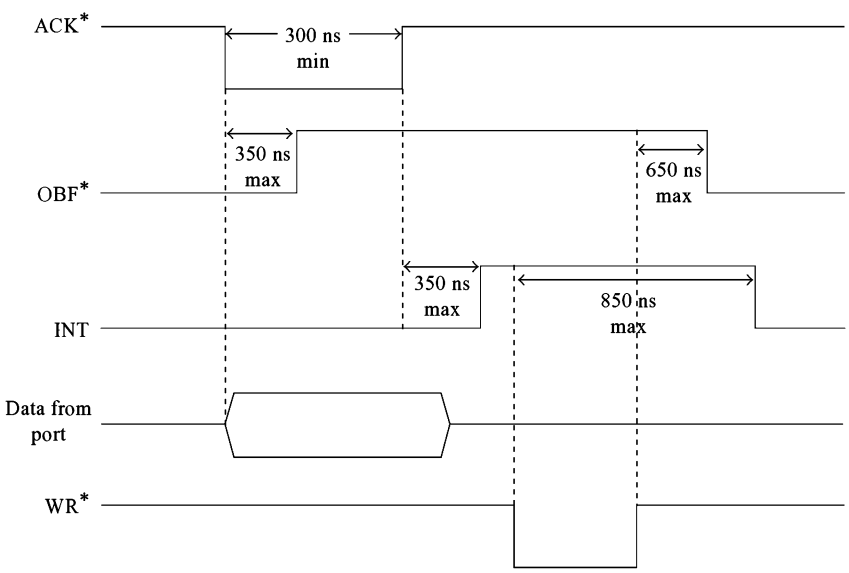


Fig. 20.13 Timing diagram for mode 1 output operation

ACK*: It is an active low ACKnowledge input pulse to the 8255. The peripheral sends a low-going pulse with a minimum width of 300 ns on this input of 8255 whenever it wants to receive data from the port. When ACK* makes logic 1 to 0 transition the latched data in the port is sent out on the port pins to the output device. For Port A, the signal is called ACK_A*, and for Port B it is called ACK_B*. This handshake pin is same as PC_6 for Port A and PC_2 for Port B.

OBF*: It is an active low output pin of 8255. It indicates output buffer full status to the peripheral and the microprocessor. The buffer is called an output buffer as it is a strobed output operation. OBF* goes to logic 0 within 650 ns after WR* goes to logic 1, indicating that the output buffer has become full. OBF* goes to logic 1 within 350 ns after ACK* makes 1 to 0 transition, to indicate that the port data is received by the output device and the buffer has become empty.

For Port A, the signal is called OBF^*_A, and for Port B it is called OBF^*_B. This handshake pin is the same as PC_7 for Port A and PC_1 for Port B.

INT: It is an active high output pin of 8255. This signal is used to interrupt the microprocessor if interrupt-driven data transfer is desired. If status check data transfer is contemplated, this pin is left open without connecting to an interrupt pin of the microprocessor. INT goes to logic 1 within 350 ns after ACK* goes to logic 1. This is so, provided that the OBF* signal is inactive and port interrupt is enabled by setting to 1 the EI flip-flop associated with the port. After a reset of 8255, Ports A and B interrupts are disabled. They are also disabled following the execution of mode definition control word.

INT goes to logic 0 within 850 ns after the WR* input signal goes to logic 0 for the purpose of writing data to the port by the microprocessor. This is true if the Port interrupt was enabled. If Port interrupt is disabled, INT output will always be in logic 0.

For Port A, the interrupt signal is called INT_A, and for Port B it is called INT_B. This handshake pin is same as PC_3 for Port A and PC_0 for Port B.

From this it should be clear that PC_2, PC_1, and PC_0 are the handshake signals for Port B when it is configured as handshake output port. Note that the same pins were the handshake signals in input mode also. Similarly, PC_7, PC_6, and PC_3 are the handshake signals for Port A when it is configured as handshake output port. Note that the handshake signals for Port A differ in input and output modes except for PC_3, which is INT_A in both input and output modes.

For the interrupt-driven output operation, the port interrupt should be enabled. If we want Port B in mode 1 output (without bothering about other ports), we have to use mode definition control as shown in the following. The chip select circuit for 8255 is assumed to be as in Fig. 20.4.

```
MVI A, 1XXXX10XB; X's are don't care bits
OUT 23H
```

Then to enable Port B interrupt, PC_2 bit is required to be set to 1 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 010 1B; X's are don't care bits
OUT 23H
```

The interrupt-driven handshake data transfer takes place as follows. Whenever the peripheral wants to receive data from the port, it senses the OBF* output of 8255. If OBF* = 0, indicating that the output buffer is full, it sends a low-going pulse on ACK*. By the time ACK* makes a 0 to 1 transition, the data in the port would have been received by the output device. By this time OBF* output is

deactivated indicating that the output buffer has become empty. Also INT output becomes logic 1. In Fig. 20.12, the 8085 is interrupted on RST5.5 pin because of INT output. It is essentially indicating to the 8085 that it has to send data to the port of 8255.

Intel 8085 finishes execution of the current instruction. Then it stores the contents of the PC on the stack top, and branches to memory location $5.5 \times 8 = 44 = 002CH$. This of course is true if interrupts are enabled, RST5.5 is unmasked, and finally RST6.5, RST7.5, and TRAP, which are the higher priority interrupts, are not active. At location 002CH there will be a jump instruction to say 2400H, where the actual ISS starts. In the ISS the 8085 executes OUT 21H, assuming Port B address to be 21H. This results in activation of CS* and WR* inputs of 8255. Also A_1 becomes 0 and A_0 becomes 1 thus selecting Port B. Thus the Port B is written by the 8085 in the ISS. Since the interrupting port is serviced, INT output of 8255 is deactivated. As the output buffer has been filled by the 8085, the OBF* output is activated. This completes the handshake data transfer by the output device in the ISS.

20.4.4 Status Check Input Operation

For this type of data transfer, the Port interrupt should be disabled. If we want Port B in mode 1 input (without bothering about other ports), we have to use mode definition control as shown in the following. The chip select circuit for 8255 is assumed to be as in Fig. 20.4.

```
MVI A, 1XXXX11XB; X's are don't care bits
OUT 23H
```

Then to disable Port B interrupt, PC_2 bit is required to be reset to 0 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 010 0B; X's are don't care bits
OUT 23H
```

In fact it does not matter even if these two instructions are not executed. This is because after execution of mode definition control the Port interrupts are automatically disabled.

In this type of data transfer the 8085 is not going to be interrupted by the 8255. So the 8085 has to perform a status check of 8255 to decide whether data transfer has to be performed or not. Port C provides status information about Ports A and B when they are in handshake mode. The details are shown in Fig. 20.14.

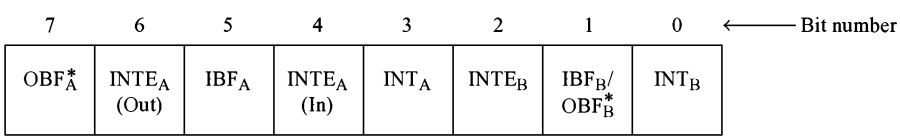


Fig. 20.14 Status information provided by Port C

Bits PC_2, PC_1, and PC_0 provide status information about Port B when Port B works in mode 1. If Port B is working in mode 0, these bits of Port C are available for use in simple I/O. Note that bit 2 indicates EI status for Port B, and not STB* or ACK* status of Port B. Among these three status bits, the most useful one is bit 1, which provides IBF_B/OBF^*_B information. If Port B is working in mode 1 input and if this bit = 1, it means that the Port B input buffer is full, and so 8085 is required to read data from Port B. Similarly, if Port B is working in mode 1 output and if this bit = 1, it means that

the Port B output buffer is empty, and so 8085 is required to send data to Port B. Thus, whenever bit 1 of Port C is 1, the 8085 is required to perform data transfer with Port B.

Bits PC_5, PC_4, and PC_3 provide status information about Port A when Port A works in mode 1 input. Then bits PC_7 and PC_6 are available for use in simple I/O. Note that bit 4 of Port C indicates EI status for Port A in input mode, and not STB* status of Port A. Among these three status bits, the most useful one is bit 5, which provides IBF_A information. If Port A is working in mode 1 input and if this bit = 1, it means that the Port A input buffer is full, and so 8085 is required to read data from Port A.

Bits PC_7, PC_6, and PC_3 provide status information about Port A when Port A works in mode 1 output. Then bits PC_5 and PC_4 are available for use in simple I/O. Note that bit 6 indicates EI status for Port A in output mode, and not ACK* status of Port A. Among these three status bits, the most useful one is bit 7, which provides OBF^*_A information. If Port A is working in mode 1 output and if this bit = 1, it means that the Port A output buffer is empty, and so 8085 is required to send data to Port A.

If Port A is working in mode 0, all the five MS bits of Port C are available for use in simple I/O.

The status check handshake data transfer in input mode takes place as follows. Whenever the peripheral has data to be sent to Port B, it senses the IBF_B output of 8255. If $IBF_B = 0$, indicating that the Port B input buffer is empty, it sends a low-going pulse on STB^*_B. When STB^*_B makes a 0 to 1 transition, the data on Port B pins are latched by the input buffer. By this time IBF_B output is activated indicating that the input buffer has become full.

However, INT_B output remains at logic 0 because Port B interrupt is disabled. Thus in this case the 8085 has to read the IBF_B status of 8255 to decide whether it is required to perform data transfer with Port B or not. If the chip select circuit for 8255 is assumed to be as in Fig. 20.4, the following program segment is executed to perform status check data transfer with Port B in mode 1 input.

```
WAIT:   IN 22H;          Read Port C to check if IBF_B has become 1.
        ANI 00000010B;   Reset all bits of Accumulator except bit 1
        JZ WAIT;         If IBF_B = 0 then wait till it becomes 1
        IN 21H;          Now read from Port B
```

20.4.5 Status Check Output Operation

For this type of data transfer, the Port interrupt should be disabled. If we want Port A in mode 1 output (without bothering about other ports), we have to use mode definition control as shown in the following. The chip select circuit for 8255 is assumed to be as in Fig. 20.4.

```
MVI A, 1010XXXXB; X's are don't care bits
OUT 23H
```

Then to disable Port A interrupt, PC_6 bit is required to be reset to 0 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 110 0B; X's are don't care bits
OUT 23H
```

In fact it does not matter even if these two instructions are not executed. This is because after execution of mode definition control the Port interrupts are automatically disabled.

The status check handshake data transfer in output mode takes place as follows. Whenever the peripheral desires to receive data from Port A, it senses the OBF^*_A output of 8255. If $OBF^*_A = 0$, indicating that the output buffer is full, it sends a low-going pulse on ACK^*_A. By the time ACK^*_A makes a 0 to 1 transition, the data in the port would have been received by the output device. By this time OBF^*_A output is deactivated indicating that the output buffer has become empty.

However, INT_A output remains at logic 0 because Port A interrupt is disabled. Thus in this case the 8085 has to read the OBF^*_A status of 8255 to decide whether it is required to perform data transfer with Port A or not. If the chip select circuit for 8255 is assumed to be as in Fig. 20.5, the following program segment is executed to perform status check data transfer with Port A in mode 1 output. The data to be sent to Port A is assumed to be at location DATA.

```
WAIT:    IN 22H;            Read Port C to check if OBF*A has become 1
         ANI 10000000B ;    Reset all bits of Accumulator except bit 7
         JZ WAIT;           If OBF*A = 0 then wait till it becomes 1
         LDA DATA
         OUT 21H;           Now send new data to Port A
```

■ 20.5 MODE 2—BI-DIRECTIONAL I/O

In mode 0 or mode 1, a port is required to work as an input port or as an output port. It depends on whether an input device or an output device is connected to the port. In contrast with this, mode 2 is called bi-directional handshake I/O. It is useful when the microprocessor sometimes desires to receive information, and at some other times desires to send information to the I/O device connected to 8255. An example is communication with a floppy disk controller card. As mode 2 is bi-directional handshake I/O, it needs more handshake lines than the uni-directional mode 1 strobed I/O. Thus, mode 2 operation makes use of five lines of Port C for handshaking purposes. These five lines of Port C are PC_7 to PC_3. For the purpose of input operation, PC_5, PC_4, and PC_3 pins provide the handshake signals for Port A when it is configured as mode 2. For the purpose of output operation, PC_7, PC_6, and PC_3 pins provide the handshake signals for Port A when it is configured as mode 2. Note that PC_3 pin will be a handshake line for Port A in mode 2 for input as well as output operations. Only Port A can work in mode 2. The remaining three lines of Port C are used for handshaking if Port B is in mode 1. However, if Port B is functioning in mode 0, these three lines of Port C are used for simple I/O.

Example 1: Configure Port A as bi-directional I/O port, Port B as basic input port, and $PC_{2\text{-}0}$ as output lines.

The required mode definition control word is shown in Fig. 20.15.

7	6	5	4	3	2	1	0	← Bit number
1	1	X	X	X	0	1	0	

Fig. 20.15 Mode definition control word for Ex. 1

Bit 0 must be a 0 to indicate that $PC_{2\text{-}0}$ are output lines. In this case it does not mean that the entire Port C lower is output, as PC_3 is used for handshaking purposes.

It is to be noted that the bit 3 is a don't care. This is because all the four lines of Port C upper are used for handshaking purposes some of which will be input pins and some others will be outputs. The 8255 will automatically configure these handshake pins, and so it is unimportant whether bit 3 is a 0 or 1.

Bit 4 is a don't care because Port A is required to work in bi-directional mode. The instructions to achieve this requirement assuming the chip select circuit of Fig. 20.4 are as follows.

```
MVI A, C2H; Treating X as 0
OUT 23H
```

Example 2: Configure Port A as bi-directional I/O port, and Port B as strobed input port. The required mode definition control word is shown in Fig. 20.16.

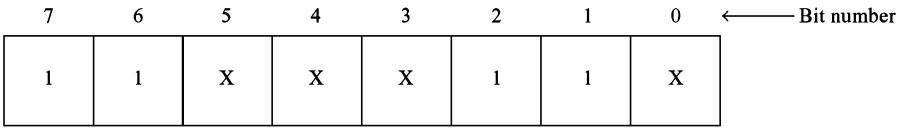


Fig. 20.16 Mode definition control word for Ex. 2

Note that bits 0 and 3 are don't cares. This is because all the eight lines of Port C are used for handshaking purposes. The instructions to achieve this requirement assuming the chip select circuit of Fig. 20.4 are as follows.

```
MVI A, FFH; Treating X as 1
OUT 23H
```

20.5.1 Interrupt-Driven Bi-Directional Operation

Let us say we have an I/O device, which supplies/receives data at irregular intervals. Then we connect it to Port A of 8255 that is configured to work in mode 2.

If we want Port A in mode 2 (without bothering about other ports), we have to use mode definition control as follows. The chip select circuit for 8255 is assumed to be as in Fig. 20.4.

```
MVI A, 11XXXXXXB; X's are don't care bits
OUT 23H
```

Then to enable Port A interrupt for input operation, PC_4 bit is required to be set to 1 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 100 1B; X's are don't care bits
OUT 23H
```

The interrupt-driven handshake data transfer from Port A to 8085 takes place as was explained earlier for mode 1 interrupt-driven input operation.

Similarly, to enable Port A interrupt for output operation, PC_6 bit is required to be set to 1 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 110 1B; X's are don't care bits
OUT 23H
```

The interrupt-driven handshake data transfer from 8085 to Port A takes place as was explained earlier for mode 1 interrupt-driven output operation.

20.5.2 Status Check Bi-Directional Operation

First of all to configure Port A in mode 2 (without bothering about other ports), we have to use mode definition control as shown below. The chip select circuit for 8255 is assumed to be as in Fig. 20.4.

```
MVI A, 11XXXXXXB; X's are don't care bits
OUT 23H
```

For this type of data transfer, Port A interrupt should be disabled. To disable Port A interrupt for input operation, PC_4 bit is required to be reset to 0 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 100 0B; X's are don't care bits
OUT 23H
```

Similarly, to disable Port A interrupt for output operation, PC_6 bit is required to be reset to 0 using Port C bit set/reset control as follows.

```
MVI A, 0 XXX 110 0B; X's are don't care bits
OUT 23H
```

In this type of data transfer, the 8085 is not going to be interrupted by the 8255. So the 8085 has to perform a status check of 8255 to decide whether data transfer has to be performed or not.

Bits PC_5, PC_4, and PC_3 provide status information about Port A in mode 2 input operation. Among these three status bits, the most useful one is bit 5, which provides IBF_A information. If this bit = 1, it means that the Port A input buffer is full, and so 8085 is required to read data from Port A.

Bits PC_7, PC_6, and PC_3 provide status information about Port A in mode 2 output operation. Among these three status bits, the most useful one is bit 7, which provides OBF^*_A information. If this bit = 1, it means that the Port A output buffer is empty, and so 8085 is required to send data to Port A.

The status check handshake data transfer in mode 2 input is performed as was explained earlier for mode 1 handshake input operation. Similarly, the status check handshake data transfer in mode 2 output is performed as was explained earlier for mode 1 handshake output operation.

[redacted]

1. With a neat diagram explain the functional pin diagram of 8255.
2. With a neat diagram explain the internal architecture of 8255.
3. Provide a chip select circuit for 8255 so that the port addresses are in the range 48H–4BH using I/O mapped I/O
4. Provide a chip select circuit for 8255 so that the port addresses are in the range 2048H–204BH using memory mapped I/O.
5. Explain mode 0, mode 1, and mode 2 operations of 8255 ports.
6. Explain mode definition control word of 8255. Write the required mode definition control word for each of the following cases.

   a. Port A: mode 0 input
      Port B: mode 0 output
      Port C upper: input
      Port C lower: output

b. Port A: mode 1 input
   Port B: mode 0 input
   Port C upper: PC_{7-6} output
   Port C lower: PC_{2-0} input

c. Port A: mode 2
   Port B: mode 1 output

d. Port A: mode 2
   Port B: mode 0 output
   Port C lower: PC_{2-0} input.

7. Explain Port C bit set/reset control word of 8255. Write the required Port C bit set/reset control word for each of the following cases.

   a. To reset to 0 bit 5 of Port C
   b. To set to 1 bit 3 of Port C

8. With a neat diagram explain the data transfer from Port B of 8255 to 8085 in interrupt-driven mode.
9. With a neat diagram explain the data transfer from 8085 to Port A of 8255 in status check mode.

21 Programs using Interface Modules

■ Description of logic controller interface
- *Evaluation of boolean expression*
- *Evaluation of boolean expression (alternative)*
- *Decimal counter using logic controller*
- *Simulation of 4-bit ALU*
- *Simulation of 8 to 1 multiplexer*

■ Successive approximation ADC interface

■ Dual slope ADC interface

■ Digital to analogue converter interface
- *Generation of rectangular wave using DAC interface*
- *Generation of triangular wave using DAC interface*

■ Stepper motor interface
- *Rotation of stepper motor in forward and reverse directions*

■ Questions

In the previous chapter 8255 PPI chip was discussed at length. In this chapter connection of some standard interface modules to ALS microprocessor kit using 8255 will be described. A number of interesting programs using these interface modules are dealt with.

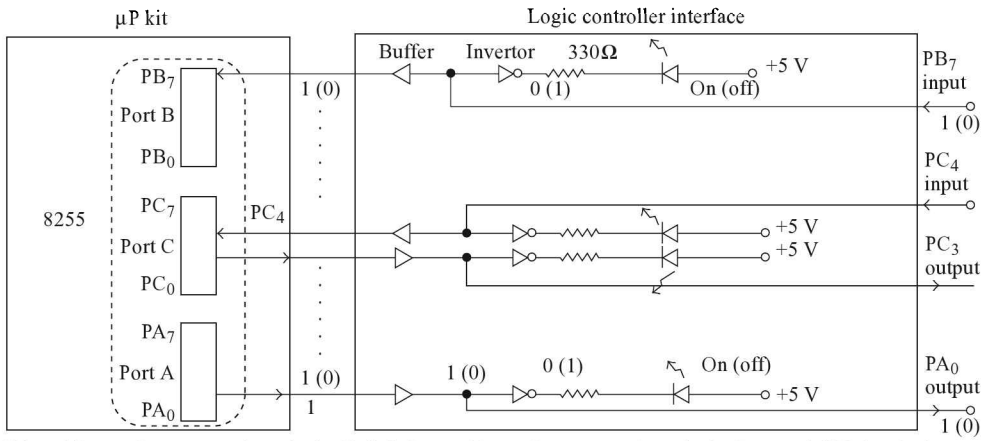
■ 21.1 DESCRIPTION OF LOGIC CONTROLLER INTERFACE

A logic controller is used in industry to control a process by software. It typically accepts multiple inputs and performs a specified sequence of arithmetic and logical operations. The multiple outputs are used to maintain the process within the desired limits. It also provides visual display of the state of the process at any instant of time.

In this section ALS-NIFC-05 model logic controller is described. It connects to ALS-SDA-85M kit using a 26-core flat cable. The connector C1 on the logic controller is connected to I/O connector P3 on the ALS kit. The 26-core flat cable connects to the connector C1 on the logic controller interface in

only the correct orientation. We say, connector C1 on the logic controller interface is polarized. Thus, we are prevented from making a wrong connection. However, care should be taken while connecting the other end of the flat cable to connector P3 (or P4) on the ALS kit. This is because P3 (or P4) connectors on the ALS kit are not polarized. Power supply of +5 V and Gnd is also connected to the logic controller. The circuit description of the logic controller is shown in Fig. 21.1a. Fig. 21.1b provides the physical layout of the interface to help the user in making the desired connections.

The logic controller interface provides 12 buffered output lines and 12 buffered input lines to the user. The 12 output lines get connected to Ports A and C (lower) of 8255 on the ALS kit. The status



Values within parantheses correspond to each other. Similarly those outside parantheses correspond to each other. For example if PB_7 input is a1, the buffer output will also be a1, the invertor output will be 0, and the LED will be On.

Fig. 21.1a Logic controller circuit details

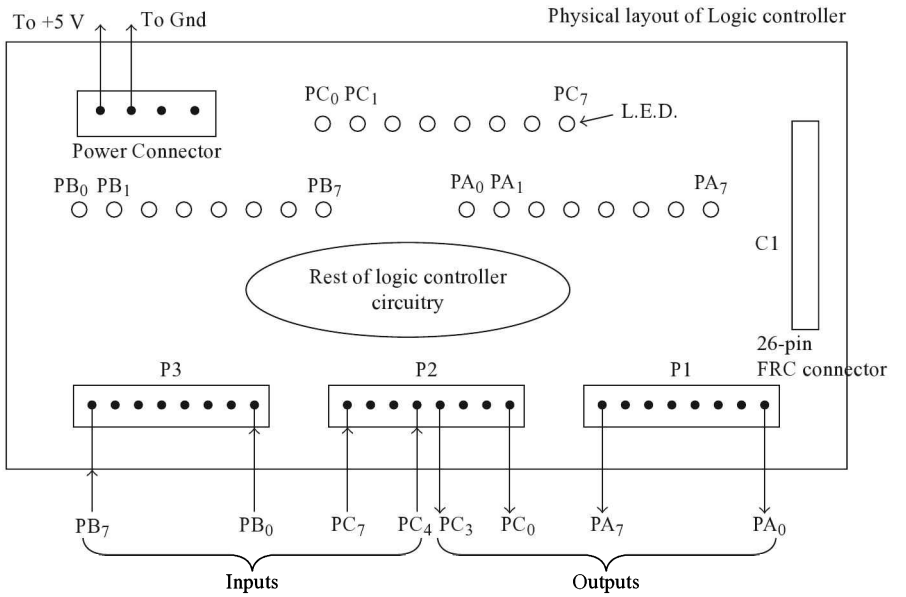


Fig. 21.1b Physical layout of the logic controller

of each of these output lines of 8255 is displayed using light emitting diodes (LED). In Fig. 21.1a the connection made to only one of the output lines of 8255 is shown.

The 12 input lines get connected to Ports B and C (upper) of 8255 on the ALS kit. The status of each of these input lines of 8255 is displayed using LEDs. In Fig. 21.1a the connection made to only one of the input lines of 8255 is shown.

When using logic controller interface with 8085 kit, configure 8255 ports on the kit as follows.

Port A as output	Port B as input
Port C (lower) as output	Port C (upper) as input

Port addresses for 8255 connected to connector P3 are as follows.

Port A: D8H Port B: D9H Port C: DAH Control: DBH

It is possible to connect the second 8255 on the ALS kit to the logic controller. For this purpose, I/O connector P4 on the ALS kit is connected to the logic controller interface, using the 26-core flat cable. Port addresses for the second 8255 on the ALS kit are:

Port A: F0H Port B: F1H Port C: F2H Control: F3H

21.1.1 Evaluation of Boolean Expression

Write an 8085 assembly language program to evaluate two 4-variable Boolean expressions, using logic controller interface.

Let us say we want to evaluate the following Boolean expressions.

$$X = P\,Q\,\overline{R}\,S + P\,\overline{Q}\,R\,\overline{S} + \overline{P}\,\overline{S} \text{ and } Y = P\,\overline{Q}\,\overline{R}\,\overline{S} + P\,\overline{R}\,S$$

First of all, truth table for the Boolean expressions is written down as shown in the following table.

P	*Q*	*R*	*S*	*X*	*Y*
0	0	0	0	1	0
0	0	0	1	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	0

PQRS inputs are connected to PB3, PB2, PB1, and PB0 of 8255 respectively. X and Y outputs are connected to PA1 and PA0, respectively. In the program that follows, the user is required to keep changing the PQRS inputs manually. The program will be in a loop outputting the X and Y values (displayed using LEDs) corresponding to the PQRS inputs.

The truth table is stored in memory as a look up table, say from location C100H. The following values (corresponding to X and Y in the LS bit positions) are stored in consecutive locations starting from C100H.

02, 00, 02, 00, 02, 00, 02, 00, 01, 01, 02, 00, 00, 03, 00, 00

Program

```
;FILE NAME BOOLEAN.ASM

       ORG C100H
TABLE  DB 02H, 00H, 02H, 00H, 02H, 00H, 02H, 00H
       DB 01H, 01H, 02H, 00H, 00H, 03H, 00H, 00H

       ORG C000H

PA     EQU D8H
PB     EQU D9H
PC     EQU DAH
CTRL   EQU DBH

       MVI A, 10001010B
       OUT CTRL      ;Configure 8255 ports

LOOP:  IN PB
       ANI 0FH       ;Now A will contain PQRS input value

       LXI H, TABLE
       ADD L
       MOV L, A      ;Point HL to proper row in the truth table

       MOV A, M
       OUT PA        ;Output XY from truth table to display
       JMP LOOP
```

21.1.2 Evaluation of Boolean Expression (Alternative)

Write an 8085 assembly language program to evaluate two 4-variable Boolean expressions, using logic controller interface. The program should test the output by automatically changing the inputs from 0000, 0001, ... to 1111 whenever a key (other than Reset and Vect Intr) is pressed.

PQRS inputs are connected to PB3, PB2, PB1, and PB0 of 8255, respectively. X and Y outputs are connected to PA1 and PA0, respectively. Program Port C lower to be the output and connect PC3-0 to the input lines PB3-0. Keep incrementing the value in Port C lower by 1 whenever a key is pressed. This results in Port B inputs getting incremented by 1, for every key depression.

Store a look up table in memory at location TABLE, as in the previous example. In the program that follows, whenever the user presses a key on the keyboard, the PQRS inputs change to the next higher value. The program will be in a loop outputting the X and Y values (displayed using LEDs) corresponding to PQRS inputs.

Program

```
;FILE NAME BOOLEAN2.ASM

       ORG C100H
TABLE  DB 02H, 00H, 02H, 00H, 02H, 00H, 02H, 00H
       DB 01H, 01H, 02H, 00H, 00H, 03H, 00H, 00H
```

```
        ORG C000H
RDKBD   EQU 0634H
PA      EQU D8H
PB      EQU D9H
PC      EQU DAH
CTRL    EQU DBH
        MVI A, 10001010B
        OUT CTRL        ;Configure 8255 ports
        MVI A, 00H
REPEAT: PUSH PSW
        OUT PC
        IN PB
        ANI 0FH         ;Now A will contain PQRS input value
        LXI H, TABLE
        ADD L
        MOV L, A        ;Point HL to proper row in truth table
        MOV A, M
        OUT PA          ;Output XY from truth table to display
        MVI A, 00001110B
        SIM             ;Unmask RST 5.5 (Keyboard interrupt)
        CALL RDKBD      ;Read a value from keyboard and discard it
        POP PSW
        INR A           ;Increment PQRS value
        JMP REPEAT
```

21.1.3 Decimal Counter using Logic Controller

Write an 8085 assembly language program to implement a decimal counter using logic controller interface. The starting count should be input through the interface and the count should be displayed on the interface.

Program

The program that follows will be in an infinite loop till the user inputs a valid two-digit BCD value to Port B. Then this initial count value is displayed by sending it to Port A. After every 0.5 s delay the count value is incremented by 1 in decimal and sent to Port A for display. Once the count value rolls over to 00 from 99, the operation repeats endlessly.

```
;FILE NAME CTR_IFC.ASM
        ORG C000H
PA      EQU D8H
PB      EQU D9H
PC      EQU DAH
CTRL    EQU DBH
DELAY   EQU 04BEH
```

```
        MVI A, 10001010B
        OUT CTRL     ;Configure 8255 ports
  ;The next 9 instructions ensure that control is transferred to next
  ;portion of program only after Port B receives a valid 2-digit BCD
input.
AGAIN:  IN PB
        ANI 0FH
        CPI 0AH
        JNC AGAIN
        IN PB
        ANI F0H
        CPI A0H
        JNC AGAIN
        IN PB
REPEAT: OUT PA      ;Display count value
        PUSH PSW
        LXI D, FFFFH
        CALL DELAY ;Generate delay of 0.5 second
        POP PSW
        ADI 01H
        DAA         ;Increment A value in decimal
        JNZ REPEAT
        JMP AGAIN
```

21.1.4 Simulation of 4-bit ALU

Write an 8085 assembly language program to simulate a 4-bit ALU using logic controller interface. The ALU should perform addition, subtraction, AND operation, or OR operation on 4-bit inputs, based on the desired operation.

The pins of the 4-bit ALU to be simulated is assumed to be as shown in Fig. 21.2.

For this ALU simulation, the inputs and outputs are provided by 8255 ports as indicated in the following.

- Pins 7-4 of Port B used as $X_{3\text{-}0}$ input;
- Pins 3-0 of Port B used as $Y_{3\text{-}0}$ input;
- Pins 7 and 6 of Port C used as S1 and S0 inputs;
- Pins 3-0 of Port A used as $Z_{3\text{-}0}$ output;
- Pin 4 of Port A is used to provide Cy output in case of add and subtract operations;
- S1 and S0 inputs determine the operation to be performed by the ALU as per the table that follows.

S1	*S0*	*Z*
0	0	X + Y
0	1	X − Y
1	0	X AND Y
1	1	X OR Y

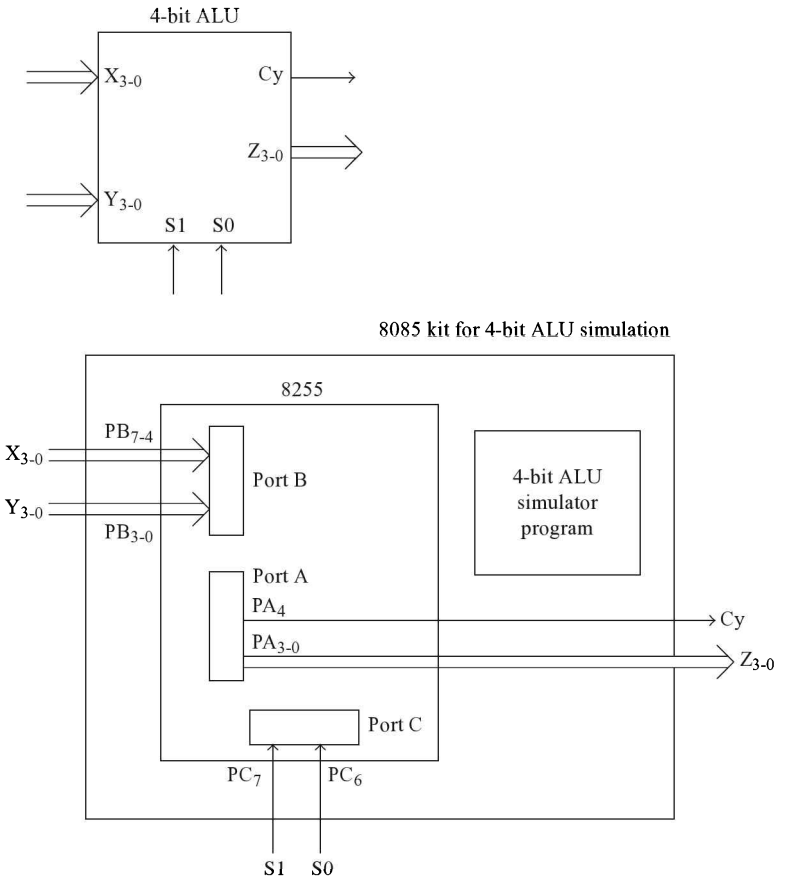


Fig. 21.2 Pin details of 4-bit ALU

Program

```
;FILE NAME ALU_IFC.ASM

       ORG C000H

PA     EQU D8H
PB     EQU D9H
PC     EQU DAH
CTRL   EQU DBH

       MVI A, 10001010B
       OUT CTRL      ;Configure 8255 ports

LOOP:  IN PB         ;Input X and Y values through Port B
       MOV B, A

       ANI 0FH
       MOV C, A      ;C will now have Y input

       MOV A, B
       ANI F0H
```

```
        RRC
        RRC
        RRC
        RRC
        MOV B, A      ;B will now have X input

        IN PC         ;Read S1 and S0 values from Port C
        ANI 11000000B

        RLC
        RLC           ;LS 2 bits of A will now have S1 and S0

        CPI 00H
        JZ ADD        ;If S1 = 0 and S0 = 0 do Add operation

        CPI 01H
        JZ SUB        ;If S1 = 0 and S0 = 1 do Subtract operation

        CPI 02H
        JZ AND        ;If S1 = 1 and S0 = 0 do AND operation

OR:     MOV A, B
        ORA C
        JMP DISP

ADD:    MOV A, B
        ADD C
        JMP DISP

SUB:    MOV A, B
        SUB C
        JMP DISP

AND:    MOV A, B
        ANA C

DISP:   OUT PA
        JMP LOOP
```

21.1.5 Simulation of 8 to 1 Multiplexer

Write an 8085 assembly language program to simulate an 8 to 1 multiplexer, using logic controller interface.

The pins of the 8 to 1 multiplexer to be simulated are assumed to be as shown in Fig. 21.3.

For this multiplexer simulation, 8255 ports as indicated in the following provide the inputs and outputs.

- Port B used as $I_{7\text{-}0}$ inputs;
- Pin 7 of Port C used as chip select;
- Pins 6-4 of Port C used as select inputs;
- Pin 0 of Port A used as output of multiplexer.

First of all, the values 01H, 02H, 04H, 08H, 10H, 20H, 40H, 80H are stored as a look up table starting at location TABLE.

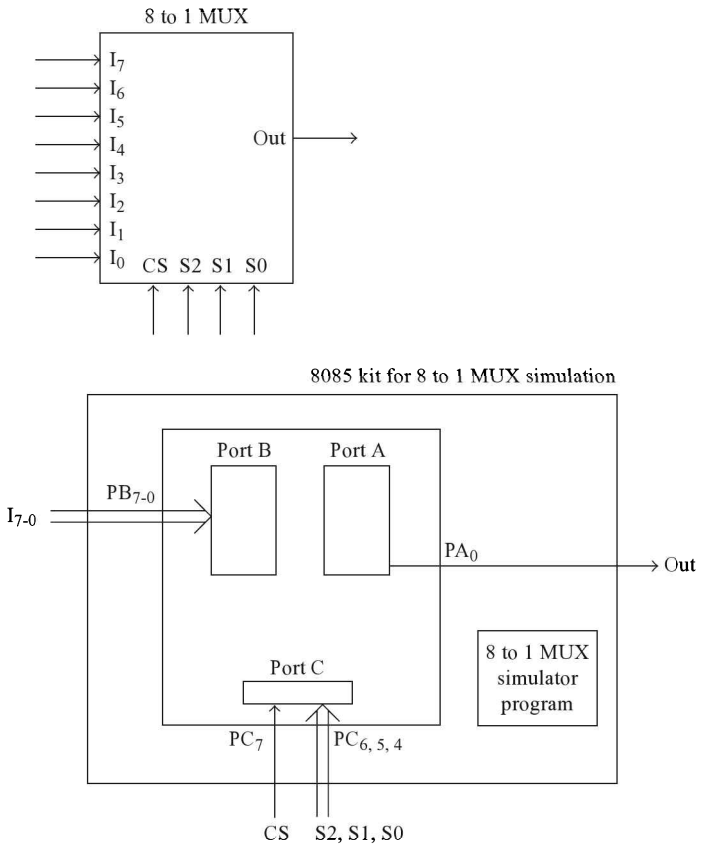


Fig. 21.3 Pin details of 8 to 1 multiplexer

Program

```
;FILE NAME MUX8TO1.ASM

       ORG C100H
TABLE  DB 01H, 02H, 04H, 08H, 10H, 20H, 40H, 80H

       ORG C000H

PA     EQU D8H
PB     EQU D9H
PC     EQU DAH
CTRL   EQU DBH

       MVI A, 10001010B
       OUT CTRL      ;Configure 8255 ports

LOOP:  IN PC
       RLC
       JNC LOOP      ;Wait in the loop till chip is enabled

       RLC
```

```
        RLC
        RLC
        ANI 07H         ;Now LS 3 bits of A will have select inputs
        LXI H, TABLE
        ADD L
        MOV L, A        ;Point HL to proper row of look up table
        IN PB
        ANA M           ;AND Port B with a value from look up table
        JZ SKIP
        MVI A, 01H
SKIP:   OUT PA          ;Send out the value of selected input
        JMP LOOP
```


■ 21.2 SUCCESSIVE APPROXIMATION ADC INTERFACE

Model ALS-NIFC-07 successive approximation ADC is described here. It also has a programmable timer interface. It connects to ALS-SDA-85M kit using a 26-core flat cable. The connector C1 on the interface is connected to the I/O connector P3 on the ALS kit, using the flat cable. Power supply of +12 V, −12 V, +5 V, and Gnd are also connected to the interface. The circuit description of the successive approximation ADC interface is shown in Fig. 21.4a. Figure 21.4b provides the physical layout details that are necessary to make the desired connections to the interface.

ADC 0809 is an 8-bit ADC. The interface uses a crystal oscillator, which provides 768 kHz as the clock input to the ADC 0809. The conversion takes 64 clocks (about 100 μs). ADC 0809 has eight analog inputs ($I_{7\text{-}0}$). One of them is selected by S2, S1, S0 inputs, which are driven by PB2, PB1, and PB0 of 8255.

The address on S2, S1, S0 is latched by ADC 0809 when ALE input becomes 1. If a '0' is sent out on PB5 of 8255, ALE input of ADC 0809 becomes a 1.

SOC (start of conversion) should become a 1, for the conversion to start. This is achieved by sending a 0 on PB6. Then the EOC (end of conversion) output of the ADC becomes a 1. When the conversion is over, EOC output is made 0 by the ADC. So, after the SOC signal is provided, we have to wait for atleast 100 μs, for the conversion to be over.

Then the digital output $D_{7\text{-}0}$, corresponding to the selected analog input, comes out on the ADC output pins if the OE (output enable) input of the ADC is at logic 1. The OE input of ADC becomes logic 1, if logic 0 is sent out on PB7. Then 8085 reads this digital value from Port A and displays it in the data field of the 8085 kit. When using this interface, it is required to configure the 8255 ports on the kit as follows.

PA as input PB as output PC as input

PC in not used by the successive approximation ADC interface. Port addresses for 8255 connected to connector P3 are as follows.

Port A: D8H Port B: D9H Port C: DAH Control: DBH

It is possible to connect the second 8255 on the ALS kit to the ADC interface. For this purpose, I/O connector P4 on the ALS kit is connected to the interface, using the 26-core flat cable. Port addresses for the second 8255 on the ALS kit are:

Port A: F0H Port B: F1H Port C: F2H Control: F3H

Problem: Write an 8085 assembly language program to interface a successive approximation ADC. Display the digital equivalent of the analog value in the data field.

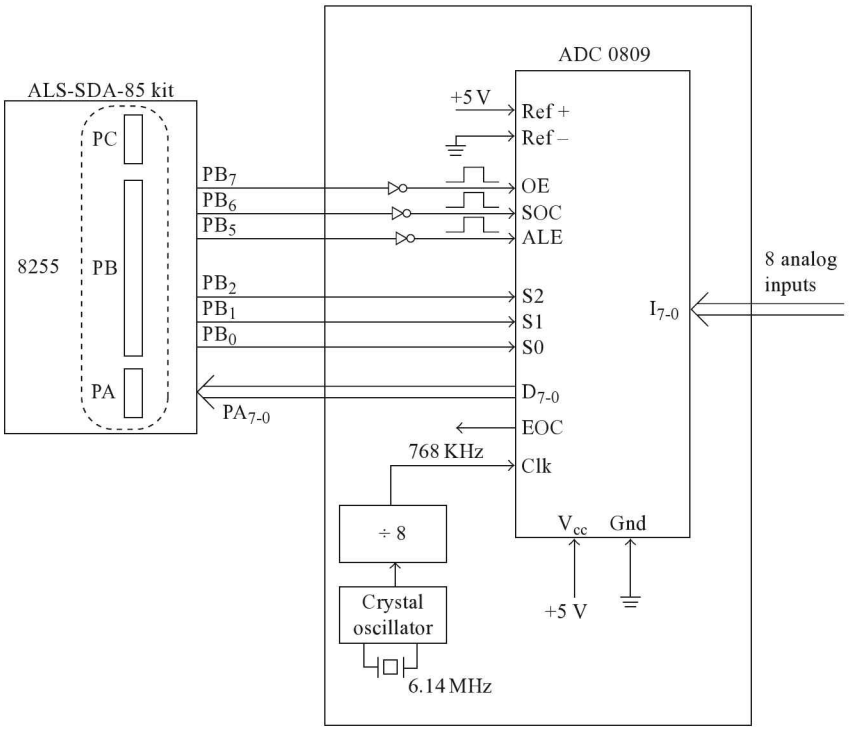
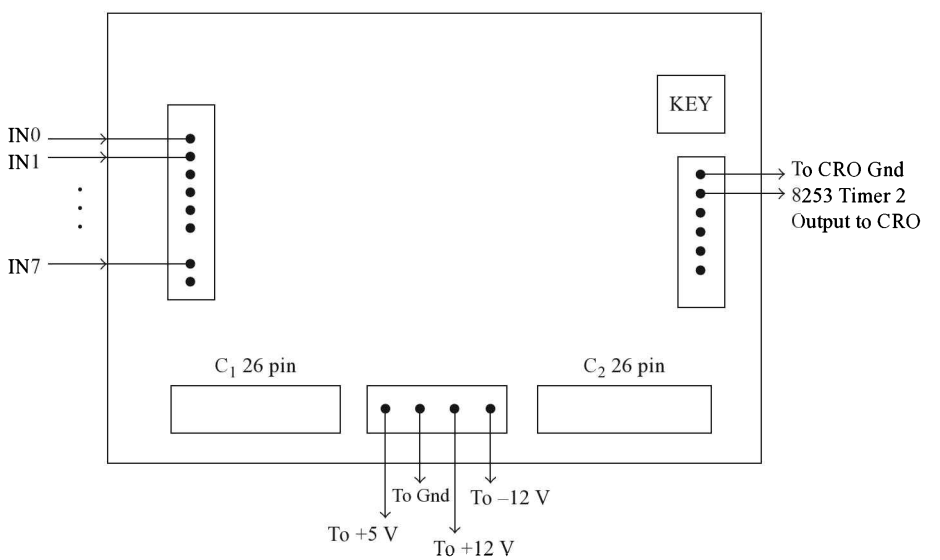


Fig. 21.4a Circuit of successive approximation ADC



C_1 is connected to connector P3 (or P4) on ALS kit, for ADC interface purpose.
C_2 is connected to connector P2 on ALS kit, for timer interface.

Fig. 21.4b Physical layout of the successive approximation ADC

Program: In this program, the user presses a key on the microprocessor kit in the range 0–7 to select an analog input.

```
;FILE NAME ADC1_IFC.ASM

       ORG C000H

PA     EQU D8H
PB     EQU D9H
PC     EQU DAH
CTRL   EQU DBH

CURDT  EQU FFF9H
UPDDT  EQU 06D3H
RDKBD  EQU 0634H
DELAY  EQU 04BEH

       MVI A, 99H
       OUT CTRL      ;Configure 8255 ports

       MVI A, 0EH
       SIM           ;Unmask RST 5.5 (keyboard interrupt)

LOOP:  EI
       CALL RDKBD

       CPI 08H
       JNC LOOP      ;Select analog input (0 to 7) for conversion

       MOV D, A      ;D now has selected analog i/p channel no.
       MVI A, 11100000B
       ADD D         ;Send to ADC selected value of analog i/p.
       OUT PB        ;Deactivate OE, SOC, and ALE
       MVI A, 10000000B
       ADD D
       OUT PB        ;Send to ADC selected value of analog i/p.
                     ;Deactivate OE. Activate SOC and ALE

       MVI A, 11100000B

       ADD D         ;Send to ADC selected value of analog i/p.
       OUT PB        ;Deactivate OE, SOC, and ALE

       LXI D, 000EH
       CALL DELAY    ;Generate delay of more than 100 microsecond

       MVI A, 01100000B
       OUT PB        ;Activate OE. Deactivate SOC and ALE

       IN PA         ;Read converted digital value from Port A
       STA CURDT
       CALL UPDDT    ;Display in data field

       JMP LOOP
```

■ 21.3 DUAL SLOPE ADC INTERFACE

Dual slope technique provides high degree of noise immunity in the conversion process. However, the conversion process is slow compared with the successive approximation ADC. Model ALS-NIFC-10 provides the user with the following.

1. DAC for ADC interface;
2. Temperature sensor interface;
3. Dual slope ADC interface.

In the discussion that follows, only dual slope ADC interface is explained. It connects to ALS-SDA-85M kit using a 26-core flat cable. The connector C1 on the interface is connected to I/O connector P3 on the ALS kit, using the flat cable. Power supply of +12 V, −12 V, +5 V, and Gnd are also connected to the interface. When using this interface, it is required to configure the 8255 ports on the kit as follows.

Port A as output	Port B as output
Port C upper as output	Port C lower as input

Port addresses for 8255 connected to connector P3 are as follows.

Port A: D8H Port B: D9H Port C: DAH Control: DBH

It is possible to connect the second 8255 on the ALS kit to the ADC interface. For this purpose, I/O connector P4 on the ALS kit is connected to the interface, using the 26-core flat cable. Port addresses for the second 8255 on the ALS kit are:

Port A: F0H Port B: F1H Port C: F2H Control: F3H

The block diagram of dual slope ADC is provided in Fig. 21.5a. The physical layout details necessary for making the desired connections to the interface is provided in Fig. 21.5b. The circuit details of dual slope ADC is provided in Fig. 21.6.

To start with, all the switches SW1, SW2, and SW3 are open. The integrating capacitor (1-μF) is then shorted for a period of about 1 μs. This is done by sending out a 1 on PB1, which results in closing on SW1. This shorts the integrating capacitor.

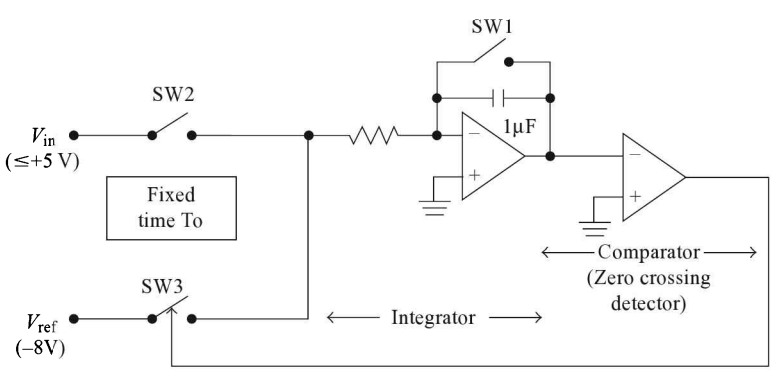


Fig. 21.5a Block diagram of dual slope ADC

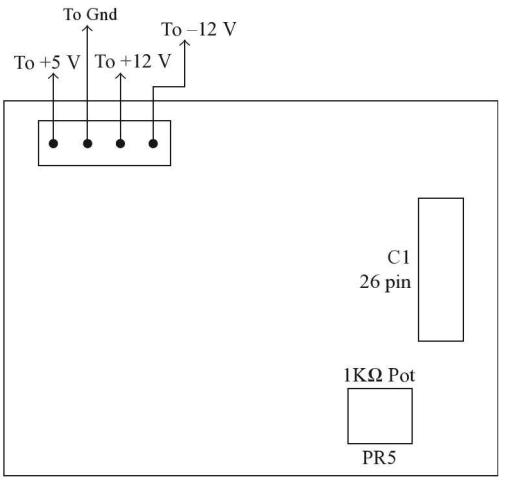


Fig. 21.5b Physical layout of the dual slope ADC interface

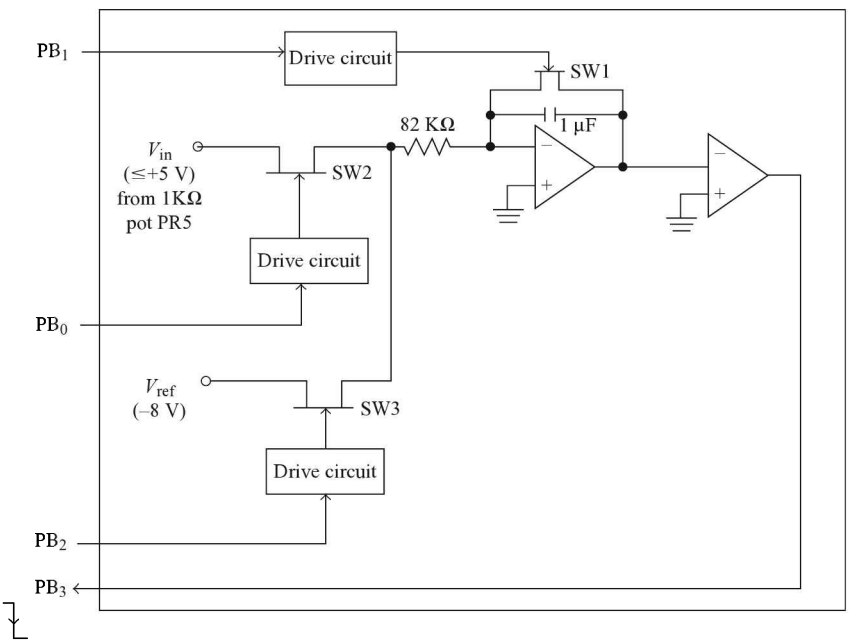


Fig. 21.6 Circuit details of dual slope ADC

The analogue input voltage V_{in} (positive voltage <= 5 V) is then integrated for a fixed time T_0. This is done by sending out an 1 on PB0, which results in closing SW2. This provides analogue input to the integrating capacitor. Now the output of the integrator reaches a negative value depending on the value of the analogue input. In the program that follows, T_0 is chosen as 4.4 μs.

A fixed reference voltage V_{ref} (negative voltage of 8 V) is then provided as input to the integrator. This is done by sending out an 1 on PB2, which results in closing SW3. This provides −8 V input to

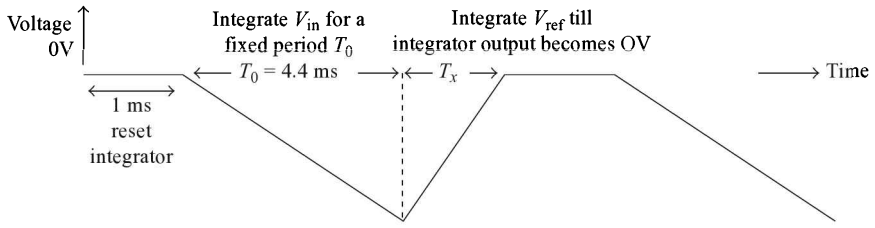


Fig. 21.7 Waveform for the integrator

the integrating capacitor. Now the output of the integrator starts moving towards 0 V from the earlier negative voltage. Let us say it takes T_x time to reach 0 V. The waveform for the integrator output is provided in Fig. 21.7.

Then V_{in} is given by the formula

$$V_{in} = \frac{V_{ref} \times T_x}{T_0}$$

After providing V_{ref} as input to the integrator, HL pair is used as an up counter in a loop till the integrator output performs zero crossing. The zero crossing is detected by the comparator. When the integrator output just goes positive, the PC3 input becomes 0. At this point of time, the value in HL pair is proportional to T_x. To be exact, T_x = contents of HL × 11 μs. This is because, a pass through the loop uses 33 clocks, each of 1/3 μs.

$$\text{Thus, } V_{in} = \frac{V_{ref} \times T_x}{T_0} = \frac{8\text{ V} \times (11\ \mu\text{s} \times \text{HL contents})}{4.4\ \mu\text{s}}$$
$$= 20\text{ mv} \times \text{contents of HL}$$

The contents of HL are displayed in the address field by the program. If V_{in} = 5 V = 5000 mV, HL pair will have 250 = 00FAH, and this value will be displayed. By slightly increasing T_0 we can get 00FF display for 5 V input.

Problem: Write an 8085 assembly language program to display in the address field the digital equivalent of the analog input, using dual slope ADC interface.

Program: The program displays the digital equivalent of the analog input in the address field. Then it waits for the user to press any key (other than Reset and Vect Intr). After a key is pressed the conversion operation starts once again and converts the latest analogue input to digital. The analogue input can be changed from 0 to 5 V using a screwdriver on the 1 KΩ potentiometer PR5 on the interface card.

```
;FILE NAME ADC2_IFC.ASM

          ORG C000H

PA        EQU D8H
PB        EQU D9H
PC        EQU DAH
CTRL      EQU DBH

CURAD     EQU FFF7H
UPDAD     EQU 06BCH
DELAY  EQU 04BEH
```

```
RDKBD  EQU 0634H

       MVI A, 81H
       OUT CTRL      ;Configure 8255 ports

       MVI A, 00H
       OUT PB        ;Switch off SW1, SW2, and SW3

BEGIN: MVI A, 02H
       OUT PB        ;Switch on SW1 to discharge capacitor

       LXI D, 0080H
       CALL DELAY    ;Generate delay to discharge capacitor

       MVI A, 01H
       OUT PB        ;Switch on SW2. Now Vin is i/p to integrator

       LXI D, 0226H
       CALL DELAY    ;Generate delay of T0 = 4.4 millisecond

       MVI A, 04H
       OUT PB        ;Switch on SW3. So Vref is i/p to integrator

       LXI H, FFFFH
AGAIN: INX H
       IN PC
       ANI 08H
       JNZ AGAIN     ;Be in loop till PC3 becomes 0. Loop needs 33 clocks.

; HL now has digital equivalent of analog i/p

       SHLD CURAD
       CALL UPDAD    ;Display HL contents in address field

       MVI A, 0EH
       SIM
       CALL RDKBD    ;Wait for a key pressing
       JMP BEGIN
```

■ 21.4 DIGITAL TO ANALOG CONVERTER INTERFACE

Model ALS-NIFC-06 dual DAC interface is described here. It also has a cassette interface, and an opto I/O interface. It connects to ALS-SDA-85M kit using a 26-core flat cable. The connector C1 on the interface is connected to I/O connector P3 on the ALS kit, using the flat cable. Power supply of +12 V, −12 V, +5 V, and Gnd are also connected to the interface.

The circuit description of the DAC portion of the interface is shown in Fig. 21.8a. The physical layout of the interface is provided in Fig. 21.8b. The DAC portion of the interface consists of two DACs, as it uses two DAC chips (DAC 0800). Our programs make use of only one of these.

DAC 0800 is an 8-bit DAC. The 8-bit digital input is fed to $DI_{7\text{-}0}$ inputs of the DAC. In the interface under consideration, Port A of 8255 on the kit provides the digital input to the DAC. Port B of 8255 on the kit provides digital input to the second DAC on the interface board.

The DAC generates analog current (not voltage) output on I_{out} pin. It is a sinking current, as shown in Fig. 21.8. Maximum current is output for the maximum digital input value of FFH. The maximum

current value will be close to I_{ref}, which in this case is 2 mA. (To be exact, the maximum I_{out} will be I_{ref} * 255/256.)

DAC 0800 provides complementing current outputs I_{out} and I_{out}*, such that always $I_{out} + I_{out}^* = I_{ref}$. This current output is converted to voltage output, using the opamp circuit shown in Fig. 21.8a. If J1 is shorted to J2, we get unipolar V_{out} as shown in the table that follows.

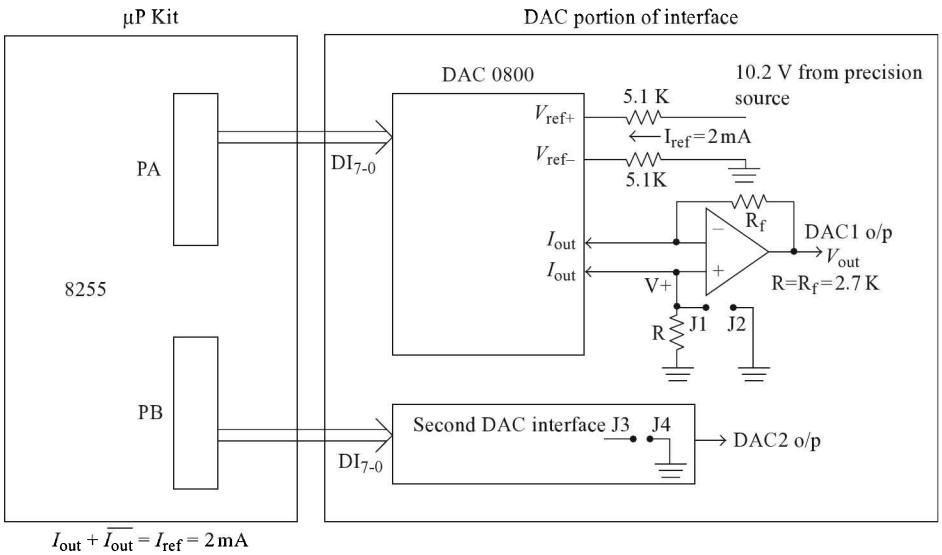


Fig. 21.8a Circuit details of DAC portion of interface

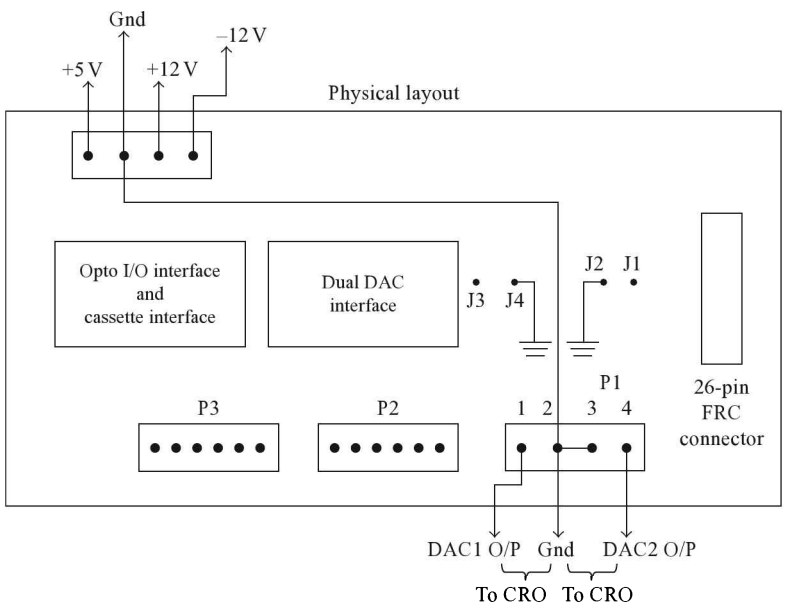


Fig. 21.8b Physical layout of the DAC interface

Unipolar Case

Digital Input	*Current I_{out}*	*$V_{out} = I_{out} \times RF$*
00H	0 mA	0 V
80H	1 mA	2.7 V
FFH	2 mA	5.4 V

If J1 is not shorted to J2, we get bipolar V_{out}, as shown in the table that follows.

Bipolar Case

Digital input	*I_{out}*	*I_{out}**	*$V_+ = -I_{out}^* \times R$*	*$V_{out} = V_+ + I_{out} \times RF$*
00H	0 mA	2 mA	−5.4 V	−5.4 V
80H	1 mA	1 mA	−2.7 V	0 V
FFH	2 mA	0 mA	0 V	+5.4 V

In this interface, the other DAC 0800 receives its digital input from Port B of 8255 on the ALS kit. Port C of 8255 connects to cassette and opto I/O interfaces. As such, when using this interface with 8085 kit, we have to configure the 8255 ports as follows.

Port A as output Port B as output
Port C Upper as input Port C Lower as output

Port addresses for 8255 connected to connector P3 are as follows.

Port A: D8H Port B: D9H Port C: DAH Control: DBH

It is possible to connect the second 8255 on the ALS kit to the ADC interface. For this purpose, I/O connector P4 on the ALS kit is connected to the interface, using the 26-core flat cable. Port addresses for the second 8255 on the ALS kit are:

Port A: F0H Port B: F1H Port C: F2H Control: F3H

In the programs using this interface, we have shorted J1 and J2, and so we get unipolar output.

21.4.1 Generation of Rectangular Wave Using DAC Interface

Write an 8085 assembly language program to generate rectangular waveform of a given duty cycle using DAC interface. Display the waveform on a cathode ray oscilloscope.

Program: To get unipolar output, J1 is shorted to J2 on the interface. To display the waveform on a CRO connect pin 1 of connector P1 to CRO signal pin, and pin 2 of connector P1 to CRO ground pin.

```
;FILE NAME DACRECT.ASM

        ORG C100H

X       DW 00FFH ;'OFF' time is proportional to this value
Y       DW 00C0H ;'ON' time is proportional to this value

        ORG C000H

PA      EQU D8H
PB      EQU D9H
PC      EQU DAH
CTRL    EQU DBH
```

```
        MVI A, 88H
        OUT CTRL      ;Configure 8255 ports

LOOP:   LHLD Y
        XCHG
        LHLD X        ;Now DE contains 00C0H and HL contains 00FFH

        MVI A, 00H
        OUT PA        ;Send 00H to DAC through Port A
        CALL DELAY    ;Generate delay proportional to HL contents

        XCHG          ;Now HL contains 00C0H
        MVI A, FFH
        OUT PA        ;Send FFH to DAC through Port A
        CALL DELAY    ;Generate delay proportional to HL contents

        JMP LOOP

;Subroutine to generate a delay proportional to contents of HL

DELAY:  DCX H
        MOV A, H
        ORA L
        JNZ DELAY
        RET
```

21.4.2 Generation of Triangular Wave Using DAC Interface

Write an 8085 assembly language program to generate triangular waveform using DAC interface. Display the waveform on a cathode ray oscilloscope.

Program: To get unipolar output, J1 is shorted to J2 on the interface. To display the waveform on a CRO, connect pin 1 of connector P1 to CRO signal pin, and pin 2 of connector P1 to CRO ground pin.

```
;FILE NAME DACTRIAN.ASM

          ORG C100H
  X       DW 00FFH  ;Rise / Fall time is proportional to this value

          ORG C000H

PA        EQU D8H
PB        EQU D9H
PC        EQU DAH
CTRL      EQU DBH

          MVI A, 88H
          OUT CTRL  ;Configure 8255 ports

     ;The next 7 instructions generate rising portion of triangular
     ;waveform. This is done by sending to DAC through Port A values from
     ;00H to FFH, in steps of 01. The increment is done after a small time
     ;delay.

LOOP:     MVI A, 00H
ASCEND:   OUT PA
          PUSH PSW
```

```
          CALL DELAY

          POP PSW
          INR A
          JNZ ASCEND

          DCR A      ;Now A contents will be FFH
;The next 7 instructions generate falling portion of triangular waveform.
;This is done by sending to DAC through Port A values from FFH to 00H,
;in steps of 01. The increment is done after a small time delay.

DESCEND:  OUT PA
          PUSH PSW

          CALL DELAY

          POP PSW
          DCR A
          CPI FFH
          JNZ DESCEND

          JMP LOOP

;Subroutine to generate delay proportional to contents of word location X

DELAY:    LHLD X
AGAIN:    DCX H
          MOV A, H
          ORA L
          JNZ AGAIN
          RET
```

■ 21.5 STEPPER MOTOR INTERFACE

ALS-NIFC-01 is a stepper motor interface. It is connected to ALS kit using 26-core flat cable. It is used for interfacing two stepper motors. In our experiment, we use only one stepper motor. The motor has a step size of 1.8°. The stepper motor works on a power supply of +12 V. Power supply of +5 V (white wire), Gnd (black), and +12 V (red) is provided to the interface. Note that −12 V supply is not used by the interface. Make sure that the +12 V supply has adequate current rating to drive the stepper motor. This is ensured by using the power supply provided with the step motor interface.

The stepper motor is connected to the interface using five-way powermate connector. The step motor is a two-phase, six-wire motor. The six wires are for the D, B, C, A inputs and VM connection (two wires). The five-way powermate connector is used for connection purpose. Ensure that red wire is connected to A1 on the interface. $PC_{3\text{-}0}$ is used to provide DBCA inputs to one stepper motor, and $PC_{7\text{-}4}$ provides DBCA inputs for the other motor, as shown in Fig. 21.9a. Thus, while using the stepper motor interface, the 8255 port C should be configured as an output port. The physical layout of the interface is provided in Fig. 21.9b.

It is possible to have four-step sequence with 'one-phase ON' scheme, as shown in the following. In this case the step size will be 1.8°.

D	**B**	**C**	**A**		
1	0	0	0	=	8
0	1	0	0	=	4
0	0	1	0	=	2
0	0	0	1	=	1

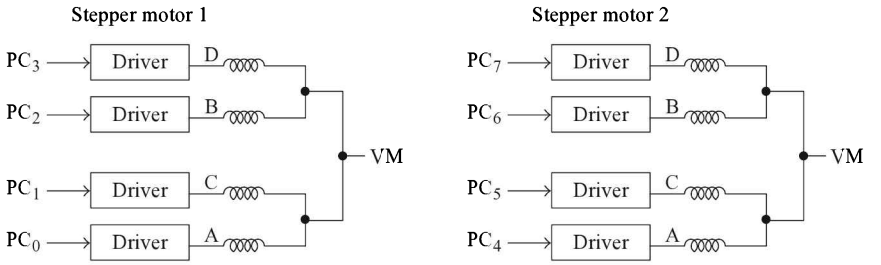


Fig. 21.9a DBCA inputs of a stepper motor

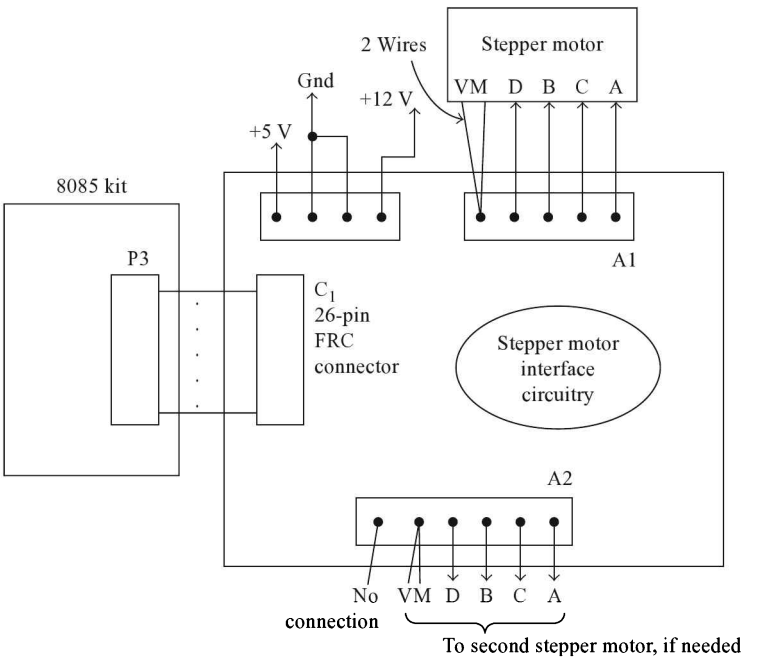


Fig. 21.9b Physical layout of the stepper motor interface

The 4 step sequence that we send to the stepper motor interface is 88H, 44H, 22H and 11H instead of 08H, 04H, 02H, 01H so that the step motor can be connected to any one of the two connectors provided on the interface board.

If the sequence is reversed, the rotation is also reversed.

21.5.1 Rotation of Stepper Motor in Forward and Reverese Directions

Write an 8085 assembly language program to rotate a stepper motor by N steps in the forward direction, then backward by the same number of steps and repeat it forever using the stepper motor controller interface.

Program: The program shown here makes the step motor rotate 100 steps of 1.8 degrees each, resulting in half revolution. then, it rotates half revolution in the opposite direction. This sequence is repeated forever. To stop the operation we have to reset the microprocessor kit.

```
;FILE NAME STEPMOTR.ASM

          ORG C100H
N         DB 100       ;100 STEPS OF 1.8° = 0.5 REVOLUTION

          ORG C000H
PA        EQU          D8H
PB        EQU          D9H
PC        EQU          DAH
CTRL      EQU          DBH
DELAY     EQU          04BEH

          MVI A, 80H
          OUT CTRL     ;CONFIGURE 8255 PORTS AS O/P IN MODE 0

BEGIN:LDA N
          MOV B, A
          MOV C, A     ;STEP COUNT VALUE IN B AND C REGISTERS

    ;THE NEXT 7 INSTRUCTIONS ARE USED FOR ROTATING BY 100 STEPS IN ONE
    DIRECTION

          MVI A, 88H;
LOOP1:OUT PC

          LXI D, FFFFH
          CALL DELAY   ;GENERATE DELAY OF 0.5 SECONDS

          RRC
          DCR B
          JNZ LOOP1

    ;NEXT 7 INSTRUCTIONS ARE USED FOR ROTATING BY 100 STEPS IN OPPOSITE
    DIRECTION

          MVI A, 88H
LOOP2:OUT PC

          LXI D, FFFFH
          CALL DELAY   ;GENERATE DELAY OF 0.5 SECONDS

          RLC
          DCR C
          JNZ LOOP2

          JMP BEGIN
```

1. With a neat diagram explain the functioning of logic controller interface. Write programs using the logic controller interface to implement the following.
   a. Decimal down counter to start from a preset value.
   b. Multiplication of 4-bit numbers and display 8-bit result.
   c. Division of 8-bit number by 4-bit number and display 4-bit quotient and 4-bit remainder. If quotient is larger than 4 bits, then overflow indication is also to be provided.
2. With a neat diagram explain the working of successive approximation ADC interface.
3. With a neat diagram explain the working of dual slope ADC interface.
4. With a neat diagram explain the working of DAC interface. Write a program using DAC interface to generate the waveform shown below.

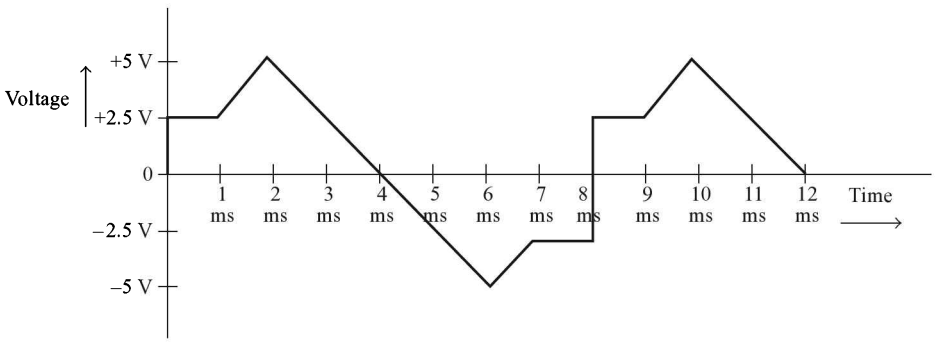


Figure of waveform

5. With a neat diagram explain the working of stepper motor interface. Write a program using stepper motor interface to rotate the stepper motor by one full revolution, then rotate by one full revolution in the opposite direction, and finally stop.

Support Chips

Chapter Heads

22 Interfacing of I/O Devices
23 Intel 8259A—Programmable Interrupt Controller
24 Intel 8257—Programmable DMA Controller
25 Intel 8253—Programmable Interval Timer
26 Intel 8251A—Universal Synchronous Asynchronous Receiver Transmitter (USART)
27 Zilog Z-80 Microprocessor
28 Motorola M6800 Microprocessor
29 8051 Microcontroller
30 Advanced Topics In 8051

INTRODUCTION

This part comprises of Chaps. 22 to 30. In the previous part of the book, 8255 PPI, which is a very popular chip that is used for interfacing I/O devices, was described in detail. Theoretically any I/O device could be interfaced with the microprocessor using the 8255. However, interfacing functionally complex I/O devices, such as keyboard and seven-segment display by using 8255 requires a lot of programming effort by the user. In such cases, it is better to use support chips that are meant specially for interfacing specific I/O devices. In this part of the book a few such popular support chips (as listed below), which are used in an 8085-based system, are described in detail.

Intel 8279—Programmable keyboard and seven-segment display controller;
Intel 8259—Programmable interrupt controller;
Intel 8257—Programmable DMA controller;
Intel 8253—Programmable timer;
Intel 8251—Programmable USART.

22 Interfacing of I/O Devices

- Interfacing seven-segment display
- Display interface using serial transfer
  - *Implementation of moving display*
- Interfacing a simple keyboard
- Interfacing a matrix keyboard
- Description of matrix keyboard interface
  - *Program to display scancode of key pressed*
- Intel 8279 keyboard and display controller
  - *Pins of 8279*
  - *Keyboard interface considerations*
  - *Interfacing seven-segment display devices*
- Programs using 8279
  - *KBD_RD routine*
  - *DISPLAY routine*
  - *XPAND routine*
  - *DATDISP routine*
  - *ADRDISP routine*
  - *Program to blank display*
  - *BLANK routine*
  - *Display scan code of key*
  - *Display characters on the kit*
  - *Use of DATDISP routine in a program*
  - *Use of ADRDISP routine in a program*
  - *Use of BLANK routine in a program*
  - *Program to change the contents of a memory location*
  - *Program to alternately display and blank*
  - *Program to check for the availability of display*
- Questions

In this chapter, a description of interfacing keyboard and 7-segment display using tri-state gates and latches is provided to begin with. Further, the advantage of using 8255 for the same purpose is explored. The problems that remain even after using 8255 are noted. Finally, the use of 8279 for interfacing keyboard and 7-segment display is described that solves all the problems encountered earlier using tristate gates, latches, or 8255 ports. In this way, the user would be in a position to appreciate the advantage of using 8279 for interfacing keyboard and 7-segment display.

■ 22.1 INTERFACING 7-SEGMENT DISPLAY

A very commonly used output device, especially in 8-bit microprocessor kits, is the 7-segment LED display. In a 7-segment LED display, there are actually eight segments, including the '.' as shown in Fig. 22.1 in the shape of character 8 with a decimal point after it. The segments are denoted as 'a, b, c, d, e, f, g, and dp' where dp stands for '.', the decimal point. Each of these segments is actually an LED or a series of LEDs. The internal circuitry of a 7-segment display is as shown in Fig. 22.2.

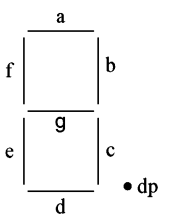


Fig. 22.1
Segments of a 7-segment display

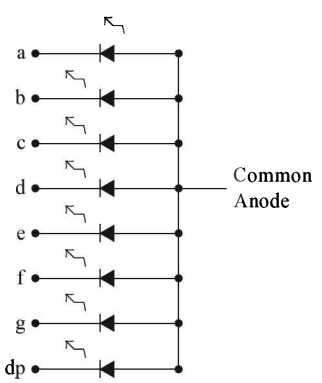


Fig. 22.2
Internal circuitry of a 7-segment common anode display

The 7-segment LED comes in two different types: the common anode type and the common cathode type. In our discussion, common anode-type 7-segment LEDs are used. In common anode 7-segment LED, the anodes of all the eight LEDs are connected together and brought out on an external pin as shown in Fig. 22.2. This pin is connected to a +5-V dc supply. The cathode ends of the eight segments are brought out on the pins of the display.

To make an LED segment glow, it is necessary to pass current through the LED segment. Typically about 10-mA current is needed to make the segment glow sufficiently bright. This is achieved by typically using 330-Ω resistor in series with an LED segment and connecting the free end of the

resistor to logic 0 state. The resistor used in this circuit is commonly called a 'current limiting resistor' for obvious reasons. The LED segment is now forward biased. There will be about 1.5-V drop across the forward-biased LED segment, and 3.3-V drop across the 330-Ω resistor when 10-mA current is passing. This works out to almost 5 V, which is the potential difference between the anode and the free end of the resistor. If more/less brightness is desired the current passing through should be increased/decreased, by decreasing/increasing the series resistance. The condition for glowing of an LED segment is shown in Fig. 22.3.

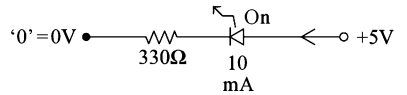


Fig. 22.3 Condition for glowing of an LED segment

If current does not pass through an LED segment, it does not glow. This is achieved by typically using 330-Ω resistor in series with an LED segment and connecting the free end of the resistor to logic 1 state. The LED segment is not forward biased now. There will be 0-V drop across the LED segment, and 0-V drop across the 330-Ω resistor as there is no current passing. This works out to 0 V, which is the potential difference between the anode and the free end of the resistor. The condition for not glowing of an LED segment is shown in Fig. 22.4.

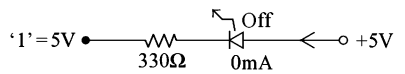


Fig. 22.4 Condition for not glowing of an LED segment

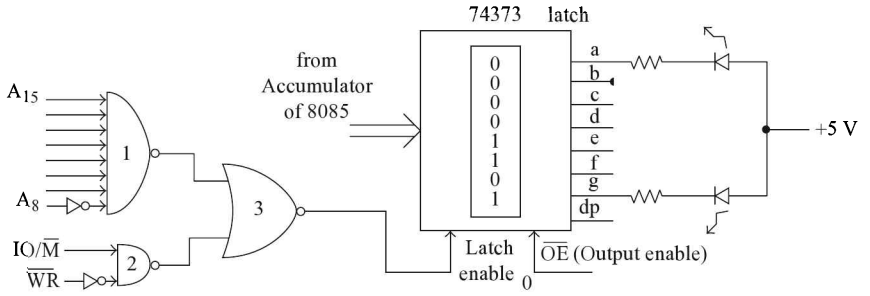
If it is desired to display the character '3' on the display, then the segments a, b, c, d, and g must be made to glow. The other segments (e, f, and dp) of the 7-segment LED must not glow. Thus the required logic state on the various inputs (free end of the resistors) of the 7-segment LED are as follows.

a	b	c	d	e	f	g	dp
0	0	0	0	1	1	0	1

= ODH

Thus if 0DH = 00001101 is input to the 7-segment display, the character '3' would be displayed. In this example, 0DH is called the 7-segment code for character '3'.

A microprocessor does not directly communicate with an output device. It communicates via an output port. *The simplest output port is basically a latch.* Suppose it is required to display the character '3' on a 7-segment display. For this, if there is no latch, the microprocessor must continuously send 0DH for the desired duration of display. It could be several seconds (several million microseconds) easily in a typical application. Thus the processor is not available for any other function for this long duration of several million microseconds. If there is a latch between the processor and the 7-segment display, the processor will send the data to the latch in a very short time (about a microsecond). Then the latch output is used to drive the 7-segment display. The processor is used very efficiently this way. The use of 74373 latch for interfacing a 7-segment display is shown in Fig. 22.5.



Note: To keep the figure simple, only two of the eight segment connections are shown. The other six segment connections are similar.

Fig. 22.5 Use of 74373 latch for interfacing a 7-segment display

In Fig. 22.5, the 74373 latch is used as an I/O mapped I/O port with the port address as FEH. This could be easily verified from the chip select circuit used in the figure. The following instructions are to be executed to display character '3' on the 7-segment display.

```
MVI A, 0DH
OUT FEH
```

When OUT FEH instruction is executed by the 8085, FEH = 11111110 is sent out on both $AD_{7\text{-}0}$ and $A_{15\text{-}8}$ during T1 of IOW machine cycle. Then gate-1 (NAND) output becomes 0. During T1, IO/M* also becomes 1, while during T2 of this machine cycle, WR* becomes 0. Hence gate-2 (NAND) output also becomes 0. This results in gate-3 (NOR) output becoming 1. This enables the latch. During T2 of this IOW machine cycle, accumulator content is sent out by the 8085 and will be present on the eight latch inputs. Thus the 74373 latches the accumulator content when OUT FEH is executed. After this, the data need not be present on the latch inputs. It is already stored in the latch. The output of 74373 is permanently enabled by connecting its output control pin to logic 0, as can be seen from the figure. So the latched data comes out on the output pins of 74373 and displays the character '3' on the 7-segment display.

More commonly, instead of using a simple latch as an output port, a port of 8255 PPI chip is used. This would result in the reduction of a number of chips, as the 8255 has three 8-bit I/O ports. Also, programmable devices are more convenient to use than non-programmable devices like 74373. The simple circuit to interface a 7-segment display as described has two main drawbacks which are indicated in the following.

In many equipments, there might be a need for several digits of display. For example, a frequency counter may need eight-digit display and eight output ports. If 74373 is used as the output port, we need eight of them. This would be quite expensive. Even if it decided to use 8255 ports, we need three chips of 8255, as each 8255 has only three ports which is quite expensive.

If character '8' is to be displayed in a digit position, all the seven segments are required to glow. As already seen, each segment when glowing draws a current of 10 mA from the power supply. Hence, if character '8' is to be displayed, the drain on the power supply would be 7×10 mA = 70 mA. If a situation arises where all the eight digits of the display are required to display the value '88888888', then the drain on the power supply is 8×70 mA = 560 mA. It is quite a large power drain indeed, especially if the equipment needs to be operated on batteries. Some equipments may have to be operated on batteries in a battlefield, where 230-V ac supply is obviously not available.

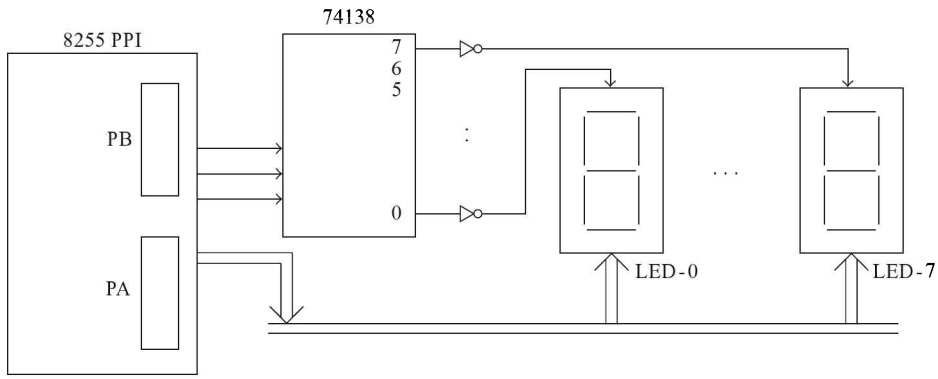


Fig. 22.6 Multiplexed display technique

The use of 'dynamic display' technique, as described in the following, could solve the problem of excessive power supply drain. It is also called 'multiplexed display' technique and the scheme is indicated in Fig. 22.6.

Let us say it is required to display '34567890' on the eight digits of the 7-segment display. First of all, the 7-segment code for character '3' is sent to all the eight LEDs for about a microsecond using Port A of 8255. For the 3 to 8 decoder the input is provided as 000 using the LS three pins of Port B of 8255. As a result the LED-0 alone gets the power supply of +5 V. Thus the character '3' is displayed only on LED-0 for about a microsecond and the other LEDs do not display anything.

Next, the 7-segment code for character '4' is sent to all the eight LEDs for about a microsecond. For the 3 to 8 decoder the input is provided as 001. As a result only LED-1 gets the power supply of +5 V. Thus the character '4' is displayed only on LED-1 for about a microsecond and the other LEDs do not display anything.

The operation is continued in a similar way for all the eight LEDs. This takes a total time of 8 μs. This constitutes one cycle of the display, in which all the required characters are displayed, but not simultaneously and continuously. If the cycles are performed one after another endlessly, the cycles are repeated (1000 μs/8 μs) = 125 times per second. This rate is higher than the persistence of vision that is about 16 times per second. As such, all the eight LEDs appear to be glowing simultaneously and continuously. The same principle is used even in the television display and cinema projections. Using this technique, even if the display is '88888888' the current drain from the power supply is only 70 mA, as only one digit is glowing at any instant. Thus the power supply drain problem is solved. However, it poses other new problems as described below.

The LED that is displaying a character should be switched off after a microsecond, and the next LED should be switched on. Once the cycle of displaying all the characters is achieved, the cycles should be repeated endlessly at a rate greater than 16 times per second. This requirement keeps the microprocessor busy most of the time for display of information. Then the processor is not much free to do processing of data, which is the most important job of the processor.

To overcome these drawbacks, a very specialized chip is used for interfacing 7-segment display units in many microprocessor-based equipments. It is the Intel 8279 display controller IC. This single chip could be used to interface up to sixteen 7-segment display units. It uses the technique of multiplexed display, but without burdening the microprocessor in this task. The details of the working of Intel 8279 are provided a little later. As would be seen later, the 8279 is also used for interfacing a matrix keyboard. As such, the 8279 chip is called 'keyboard and display controller'.

■ 22.2 DISPLAY INTERFACE USING SERIAL TRANSFER

In this section, interfacing four numbers of 7-segment LEDs using serial data transfer is explained. The display portion of the interface has four 7-segment LEDs as can be seen from the physical layout of the interface provided in Fig. 22.7. This figure is of help to the user in making the required connections. The interface connects to the ALS-8085 kit using a 26-core flat cable. The connector C1 on the interface is connected to I/O connector P3 on the ALS kit. A power supply of +5 V and Gnd is also connected to the interface.

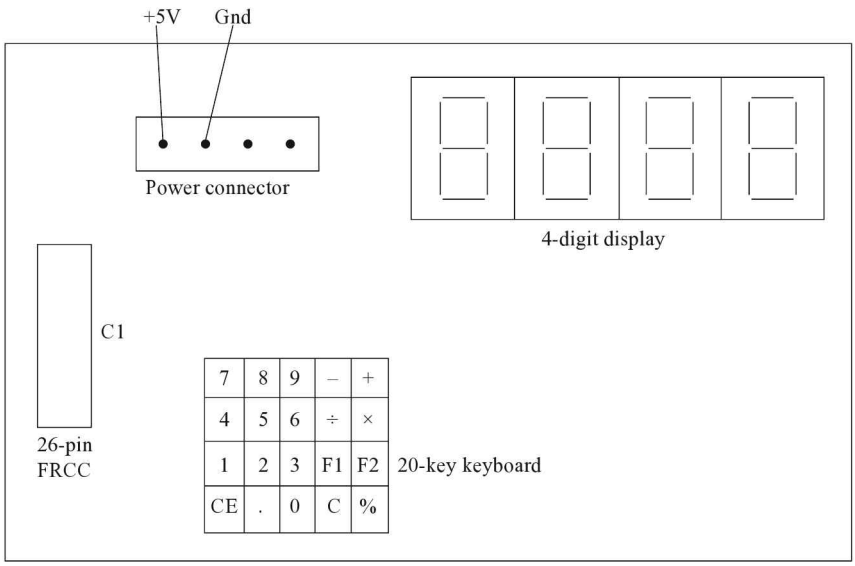


Fig. 22.7 Physical layout of display and keyboard interface

The interface uses four 8-bit serial shift registers (type no. 74164). The outputs of these shift registers are connected to the segments of the LEDs as shown in Fig. 22.8 that provides circuit details of the display interface. PC_7, PC_6, PC_5, and PC_4 are used to provide clock pulses to the four shift registers. The bit value on PB_0 enters segment 'a' of the MS digit, when the clock makes a 0 to 1 transition. When another clock transition occurs, the data at segment 'a' is shifted to segment 'b' and the new bit value at PB_0 enters segment 'a'. Proceeding this way, after eight clock transitions, a character is formed in the MS digit position. After eight more clock transitions, the character at MS digit position is shifted right to the next digit position. Thus, after $8 \times 4 = 32$ clock pulses, four digits are displayed on the interface.

When using this interface it is necessary to configure the 8255 ports as follows.

Port B as output;
Port C upper portion as output;
Configuration of other ports is unimportant.

Port addresses for 8255 connected to connector P3 are as follows:

Port A: D8H Port B: D9H Port C: DAH Control: DBH

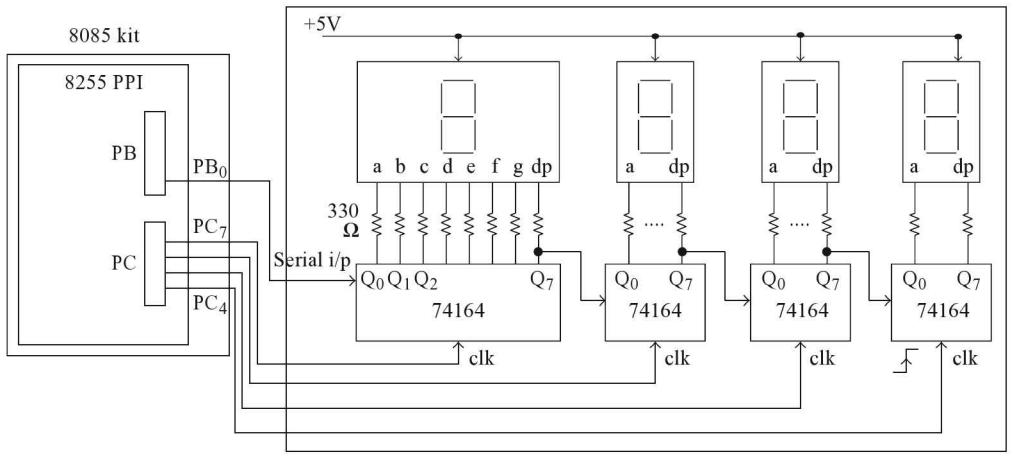


Fig. 22.8 Circuit details of display interface

It is possible to connect the second 8255 on the ALS kit to this interface. For this purpose, the I/O connector P4 on the ALS kit is connected to this interface using the 26-core flat cable. Port addresses for the second 8255 on the ALS kit are:

Port A: F0H Port B: F1H Port C: F2H Control: F3H

22.2.1 Implementation of Moving Display

Write an 8085 assembly language program to implement a moving display of a given string of digits on the display interface. The display content should move by a one-digit position to the right every second.

The program which follows displays the characters '0' to '9' in a moving display fashion. The characters to be displayed on the interface in consecutive seconds are indicated as follows.

x x x x	Any junk character is denoted as 'x'
0 x x x	
1 0 x x	
2 1 0 x	
3 2 1 0	
4 3 2 1	
5 4 3 2	
6 5 4 3	
7 6 5 4	
8 7 6 5	
9 8 7 6	
b 9 8 7	Blank character is denoted as 'b'
b b 9 8	
b b b 9	

b b b b Above sequence of length 14 repeats from now on
0 b b b
1 0 b b
2 1 0 b
3 2 1 0

First of all, the user is required to store in consecutive memory locations the 7-segment codes for the characters to be displayed on the interface. The 7-segment codes for the various digits are as follows.

'0' = 03H	'1' = 9FH	'2' = 25H	'3' = 0DH	'4' = 99H
'5' = 49H	'6' = 41H	'7' = 1FH	'8' = 01H	'9' = 09H

The 7-segment codes for any character can easily be derived. For example, the derivation of 7-segment code for '3' is reproduced as follows from earlier discussion.

a	b	c	d	e	f	g	dp
0	0	0	0	1	1	0	1 = 0DH

After storing the above mentioned ten 7-segment codes, the 7-segment code for blank (FFH) is stored in the next four memory locations. Thus, the 7-segment codes for the 14 characters to be displayed on the interface in moving display fashion are stored in consecutive memory locations.

Program

```
          ORG C100H
TABLE     DB 03H, 9FH, 25H, 0DH, 99H, 49H, 41H
          DB 1FH, 01H, 09H, FFH, FFH, FFH, FFH

          ORG C000H
PA        EQU  D8H
PB        EQU  D9H
PC        EQU  DAH
CTRL      EQU  DBH

DELAY     EQU  04BEH

          MVI A, 10000000B
          OUT CTRL        ;Configure PB and PC upper as output ports

AGAIN:    LXI H, TABLE    ;HL points to TABLE
          MVI B, 0EH      ;(B) = 14, the no. of characters in the sequence
NXTCHAR:  MVI E, 08H      ;(E) = 08, the no. of segments per character
          MOV A,M         ;7-segment code for the character to display
                          ;moved to A

NXTSEG:   OUT PB          ;Send segment value on PB0 to serial input of
                          ;shift register
          MOV D,A         ;Save A value in D

          MVI A, 00H
          OUT PC          ;Send 0000 on PC7-4
          MVI A, F0H
          OUT PC          ;Send 1111 on PC7-4. Generate 0 to 1 transition
                          ;for clock
          DCR E           ;Decrement segment counter
          JZ OVERCHK      ;If the character display over, check if it was
                          ;last character
```

```
          MOV A, D        ;Reload A from D
          RRC             ;Load LS bit of A with new segment value
          JMP NXTSEG      ;Go to display next segment of character
OVERCHK:  CALL DLY1SEC    ;Generate delay of 1 sec after a character is
                          ;formed
          DCR B           ;Decrement character counter
          JZ AGAIN        ;Repeat the sequence if all characters already
                          ;displayed
          INX H           ;Point HL to next character code in the sequence
          JMP NXTCHAR     ;Go to display next character in the sequence

  ;Subroutine to generate delay of 1 second. This subroutine is executed
  ;after a character is fully formed. This results in display of the
  ;character for a full second before it is moved to next position.
DLY1SEC:  MVI C, 02H      ;Counter that generates delay in steps of 0.5 seconds
          PUSH D          ;Save DE
          LXI D, FFFFH    ;
REP:      CALL DELAY      ;Generate delay of 0.5 second
          DCR C           ;Decrement C after a delay of 0.5 second
          JNZ REP         ;If C is non zero, generate delay of another 0.5 second
          POP D           ;Restore DE
          RET
```

■ 22.3 INTERFACING A SIMPLE KEYBOARD

A very commonly used input device is the keyboard. Let us say there are only eight keys on the simple keyboard. Whenever the user presses a key on this keyboard, the microprocessor should immediately identify the key pressed. The action to be performed by the processor depends on the key that is pressed. For example, on a calculator keyboard, addition operation is to be performed when '+' key is pressed.

A microprocessor does not directly communicate with an input device. It does it via an input port. *The simplest input port is basically a set of tristate gates.* A commonly used tri-state buffer is the 74244 chip. It has totally eight buffers arranged as two groups of four buffers with non-inverted tri-state outputs as shown in Fig. 22.9.

The 4-bit input comes out on the corresponding outputs of the 74244 only when the enable pin for the 4-bit group is at logic 0. It is like having a short circuit between the input and the output when the enable pin is at logic 0. If the enable pin is at logic 1, the input does not come out on the corresponding output and is blocked from reaching the output. It is like having an open circuit between the input and the output when the enable pin is at logic 1. In such a case, the outputs are said to be in high impedance state, also called as tristate. The two active low enable pins are connected together when the chip is to be used as an octal tristate buffer.

In Fig. 22.9, the 74244-tristate buffer is used as an I/O-mapped I/O port with the port address as 77H. This could be easily verified from the chip select circuit used in the figure. When IN 77H instruction is executed by the 8085, 77H = 01110111 is sent out on both $AD_{7\text{-}0}$ and $A_{15\text{-}8}$ during T1 of IOR machine

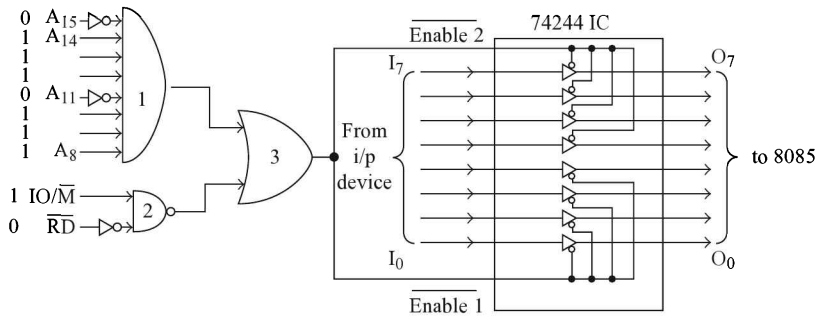


Fig. 22.9
Octal tri-state buffer chip—74244

cycle. Then gate-1 (NAND) output becomes 0. During T1, IO/M* also becomes 1. During T2 of this machine cycle, RD* becomes 0. Hence gate-2 (NAND) output also becomes 0 which results in gate-3 (OR) output becoming 0. This enables the tristate buffer. During T2 of this IOR machine cycle, the data presented by the input device is received in the accumulator of 8085 via the data bus. Thus the 74244-tristate buffer content is received in the accumulator of 8085 when IN 77H is executed.

In a microcomputer system there could be a number of input devices. The processor would like to select one of them and read the data present in the input device. All the input devices send their data to the data bus using tristate gates as shown in Fig. 22.10.

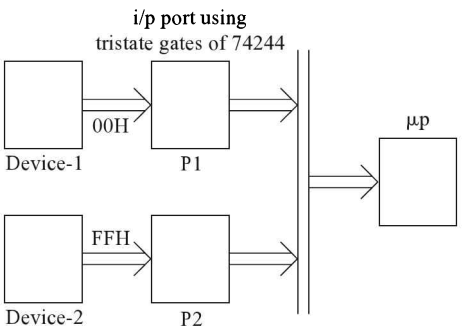


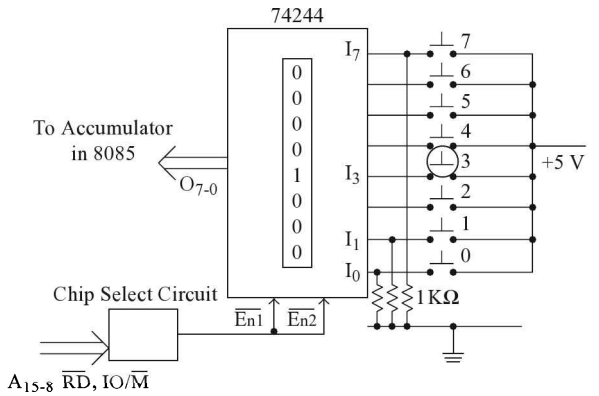
Fig. 22.10
Tristate gates as input ports

Suppose the input devices directly send their data to the data bus without the intervening tristate buffers. Also suppose that device-1 and device-2 send the data 00H and FFH respectively. Then the data bus is attempted to be loaded with 00H and FFH simultaneously. This will damage the microcomputer system. As per the scheme shown in Fig. 22.10, the data bus is loaded with 00H if the processor selects P1 and FFH if it selects P2.

More commonly, instead of using a simple tristate buffer as an input port, a port of 8255 PPI chip is used. This would result in a reduction of the number of chips, as the 8255 has three 8-bit I/O ports. Also, programmable devices are more convenient to use than non-programmable devices like 74244.

A circuit that uses 74244 to interface a simple keyboard with eight keys is shown in Fig. 22.11.

In Fig. 22.11, one end of all the keys is connected to logic 1. The other end of the key that is connected to the 74244 input is also connected to ground through a resistor. This resistor is normally called a pull-down resistor. A pull-down resistor helps in bringing the logic level at a key that is not pressed to logic 0. Pull-down resistor value used is typically 1 KΩ. If only key 3 is pressed, then I3 input of 74244 will be logic 1. In this case the pull-down resistor will have no effect in the circuit operation. All other inputs of 74244 will be logic 0. The pull-down resistors help in bringing these inputs of 74244 to logic 0. Thus, when only key 3 is pressed, the inputs to the 74244 would be 00001000 = 08H. This value gets loaded into the accumulator when IN 77H instruction is executed.



Note:
For simplicity, pull down resistors are shown for I_0, I_1 and I_7 inputs only. However, they are present for the remaining inputs also.

Fig. 22.11 Interfacing a simple keyboard using 74244-tristate buffer

The circuit to interface a simple keyboard as described earlier has several drawbacks. They are indicated as follows.

- Assume that when IN77H instruction is executed, the accumulator receives the value 00000000 = 00H. It means that none of the eight keys are pressed at that moment. But the user expects the processor to detect and identify the key as soon as it is pressed. For this reason, the processor should keep checking in a loop till a key is pressed, as indicated below.

```
AGAIN:  IN 77H
        CPI 00H
        JZ AGAIN
```

- The processor exits the loop only when a key is pressed. Let us say the user presses a key on the keyboard once every second on an average, which means that the processor spends 1 s in the loop. In 1 s, the processor could have executed several hundred thousand useful instructions. Instead of that, it is simply wasting its time in the loop. This is a very inefficient way to utilize the capabilities of the processor. To solve this problem, a mechanism should be provided that interrupts the processor whenever a key is pressed. Only then the processor should read the input port. This would avoid the waste of time by the processor in the loop waiting for a key to be pressed.
- The keys used in the keyboard are spring loaded mechanical keys. So when a key is pressed, it performs a number of 'makes' and a number of 'breaks', before it settles down to make the contact. Similarly, when the finger is removed from a pressed key, it performs a number of 'breaks' and a number of 'makes', before it settles down to break the contact. This problem is called 'contact bouncing'. The design of the keyboard is such that the bouncing dies down in about 20 μs. Thus, when a key is found to be pressed, provision has to be made to check it again after a debouncing time of about 20 μs, to make sure that the key is really pressed.
- If the number of keys is more, say 64, then eight numbers of 74244 chips are needed, which increases the cost. Even if it is decided to use 8255 ports, three 8255 chips are needed, as each 8255 has only three ports.

Using a matrix keyboard solves the last problem. Even if there are 64 keys, just one 8-bit output port and one 8-bit input port are enough for a matrix keyboard. The interfacing of matrix keyboard is described next.

■ 22.4 INTERFACING A MATRIX KEYBOARD

A matrix keyboard will have keys arranged in the form of a matrix of several rows and columns. Figure 22.12 indicates the interfacing of a matrix keyboard having four rows and four columns. At the intersection of every row and column a key is connected. Thus there are a total of $4 \times 4 = 16$ keys in the matrix. The column lines are connected to Gnd through pull-down resistors.

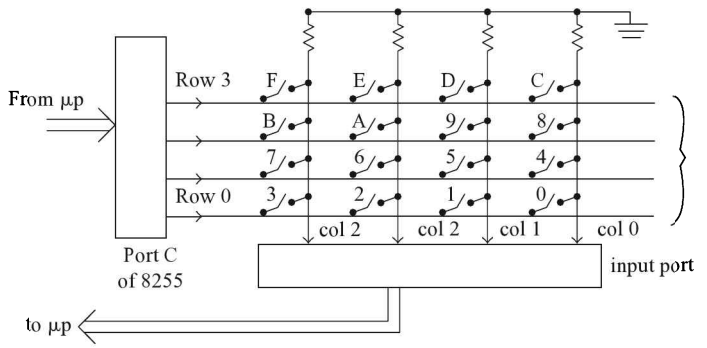


Fig. 22.12
Interfacing a matrix keyboard

Even if the matrix size were to be 8×8, for a total of 64 keys, just one output port and one input port would suffice. In the case of a simple keyboard discussed earlier, eight input ports were needed for this size of the keyboard. The identification of the key pressed on the keyboard takes place as follows.

For example, let us say the user presses key 9 that is at the intersection of row 2 and column 1. The microprocessor performs a scan of the matrix keyboard, a row at a time. It starts this operation with row 0. Using the output port the processor sends logic 1 on row 0. On the other rows, logic 0 is sent out. This is done very easily using the instructions

```
MVI A, 00000001B
OUT PC
```

Then the processor reads the input port. Presently, logic 1 is sent out only on row 0. But no key is pressed on this row. As such, the input port receives the value 0000 from the keyboard because of the pull-down resistors. The reader might think about the key 9 that is pressed on row 2. But logic 0 is sent out presently on row 2. Hence only logic 0 is received on column 1 via the pressed key. The processor compares the value that is read from the input port with 0000. If both are same, it means that no key is pressed on row 0.

The processor then scans row 1. For this purpose, using the output port the processor sends logic 1 on row 1, while logic 0 is sent out on the other rows. The actual programming details are omitted at this stage for simplicity. Then the processor reads the input port. Presently, logic 1 is sent out only on row 1. But no key is pressed on this row. As such, the input port receives the value 0000 from the keyboard because of the pull-down resistors. The reader might think about the key 9 that is pressed on row 2. But logic 0 is sent out presently on this row. Hence only logic 0 is received on column 1 via the pressed key. The processor compares the value that is read from the input port with 0000. If both are the same, it means that no key is pressed on row 1 also.

The processor scans row 2 next. For this purpose, the processor sends logic 1 on this row, while logic 0 is sent out on the other rows. Then the processor reads the input port. Now because of the logic 1 present on row 2, and the pressing of key 9, the input port receives 0010, which means logic 1 on

column 1 and logic 0 on the other columns. The processor compares the value that is read from the input port with 0000. As they are not same, it means that a key is pressed. The pressed key is present on row 2, as this row was presently scanned. The pressed key is present on column 1, as the input port received logic 1 from column 1. Thus it means that the key pressed was at the intersection of row 2 and column 1.

This row and column information about the key pressed must be converted by the software to the value 9 so that the processor identifies that key 9 was pressed. In this example, multiplying the row number by 4 (as there are four keys in every row) and adding the column number as shown in the following achieves the objective.

$4 \times$ row no. + col. no.
$4 \times 2 + 1 = 9$

The advantage of matrix keyboard is that only one input port and one output port are enough even when there are eight rows and eight columns, for a total of 64 keys in the matrix.

The disadvantages of the matrix keyboard are as follows.

- Suppose after scanning all the rows it was found that no key was pressed by the user. But the user expects the processor to detect and identify the key as soon as it is pressed. For this reason, the processor should keep scanning the entire keyboard in an infinite loop till a key is pressed. This is obviously a very inefficient way to utilize the capabilities of the processor. To solve this problem, a mechanism should be provided that interrupts the processor whenever a key is pressed. Only then the processor should perform the scanning. This would avoid the waste of time by the processor in the scanning process.
- The contact bounce problem still persists in the matrix keyboard. Thus, when a key is found to be pressed, a provision has to be made to check it again after a debouncing time of about 20 μs, to make sure that the key is really pressed.
- The processor has to execute a lengthy program to scan the matrix keyboard. This is because the processor has to execute a number of instructions to scan the keyboard a row at a time. When a key is found pressed, it has to confirm the pressing of the key after debouncing. Finally it has to convert the row number and column number to proper key value.

To overcome these drawbacks, a very specialized chip is used for interfacing matrix keyboards in many microprocessor-based equipments. It is the Intel 8279 keyboard and display controller IC. This single chip can be used to interface an 8×8 matrix keyboard. This chip automatically scans a row at a time. If a key is found pressed, it will wait for the de-bounce time, and then again scans the keyboard. Only after debouncing the keyboard, it interrupts the processor if it finds a key pressed. In the interrupt service routine the processor reads from this chip to find out about the key pressed. In this way, the processor is completely freed from the problem of scanning the keyboard, wait for de-bounce time, and the like. The details of the working of Intel 8279 are provided a little later.

■ 22.5 DESCRIPTION OF MATRIX KEYBOARD INTERFACE

The purpose of this section is to acquaint the reader with the programming complexity involved in interfacing a matrix keyboard using conventional I/O ports of 8255 PPI. This would help the reader appreciate the benefits derived from using 8279 chip as described later.

For this purpose, readily available keyboard and display interface (Model ALS-NIFC-09) from M/s Advanced Electronics Systems is made use of. This interface has two parts: keyboard interface and display interface. In this section only the keyboard interface portion is explained. The display interface has already been explained in an earlier section.

The keyboard interface contains 20 keys. They are 0, 1,..., 9, '.', '−', '+', '÷', '×', '%', C, CE, F1, and F2. The physical layout in Fig. 22.7 gives the impression that it is a matrix keyboard with four rows and five columns. The circuit details of the keyboard interface portion are provided in Fig. 22.13. From this figure it should be clear that the keys are arranged as a matrix of three rows and eight columns. As such, 24 keys could have actually been there on the keyboard. However, as the keyboard is intended for a simple calculator, the 20 keys provided on the keyboard is adequate.

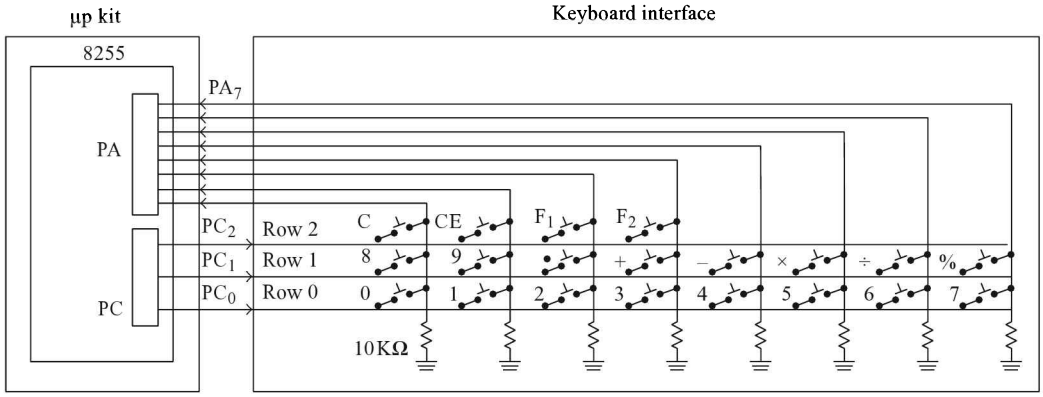


Fig. 22.13 Circuit details of the keyboard interface

The interface connects to the ALS-8085 kit using a 26-core flat cable. The connector C1 on the interface is connected to I/O connector P3 on the ALS kit. A power supply of +5 V and Gnd are also connected to the interface.

The rows are connected to PC_2, PC_1, and PC_0. Port C lower is configured to work as output port. The columns are connected to $PA_{7\text{-}0}$. Port A is configured to work as input port. Thus, when using this interface it is necessary to configure the 8255 ports as follows.

Port A as input;
Port C lower portion as output;
Configuration of other ports is unimportant.

Port addresses for 8255 connected to connector P3 are as follows:

Port A: D8H Port B: D9H Port C: DAH Control: DBH

It is possible to connect the second 8255 on the ALS kit to this interface. For this purpose, the I/O connector P4 on the ALS kit is connected to this interface using the 26-core flat cable. Port addresses for the second 8255 on the ALS kit are:

Port A: F0H Port B: F1H Port C: F2H Control: F3H

22.5.1 Program to Display Scancode of Key Pressed

Write an 8085 assembly language program that uses 3 × 8 matrix keyboard interface, and display in the data field of the kit the scan code of the key that is pressed on the interface.

The scan codes assigned for the various keys on the keyboard interface are as follows.

Key	*Scan code*
0	00
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	08
9	09
.	0A
−	0B
+	0C
÷	0D
×	0E
%	0F
C	10
CE	11
F1	12
F2	13

In the 3 × 8 matrix keyboard, the scan code of the key pressed is obtained by multiplying the row number by 8 (as there are eight keys in every row) and adding the column number.

Program

```
          ORG C000H
PA        EQU  D8H
PB        EQU  D9H
PC        EQU  DAH
CTRL      EQU  DBH

UPDDT     EQU  06D3H
CURDT     EQU  FFF9H

          MVI A, 10010000B
          OUT CTRL      ;Configure PA as input and PC lower as output port

AGAIN:    CALL SCANKBD ;Get the scan code of key pressed in A register

          MVI B, 00H
          STA CURDT
          CALL UPDDT    ;Display scan code in Data field with no dot

          JMP AGAIN     ;Remain in infinite loop

SCANKBD:  MVI C, 03H    ;C indicates the no. of rows still to scan
          MVI D, 00H    ;D indicates the row number being scanned
          MVI A, 80H

NXTROW:   RLC           ;Now, A has required pattern to scan desired row
          MOV B, A      ;Save A value in B
          OUT PC        ;Send logic 1 on the row to be scanned
```

```
            IN PA          ;Read the columns of the keyboard
            CPI 00H        ;Compare with 00H
            JNZ KEY_ID     ;If a key is pressed on the row scanned go to KEY_ID

            MOV A, B       ;Restore A value
            INR D          ;D points to next row
            DCR C          ;
            JNZ NXTROW     ;Go to NXTROW if scan of all rows not complete
            JMP SCANKBD    ;Be in an infinite loop if no key is pressed
KEY_ID:     MVI C, 08H     ;C used as down counter. No. of columns is 8
            MVI E, 00H     ;E indicates the column being checked

REPEAT:     RRC            ;Logic state of column E brought to Carry
            JC SKIP        ;Go to SKIP if logic state of column E is 1

            INR E          ;E points to next column
            DCR C;
            JNZ REPEAT     ;Go to REPEAT if testing of all columns not over

SKIP:       MOV A, D       ;Get the row number in A.
            RLC
            RLC
            RLC            ;Multiply row number by 8
            ADD E          ;Add the column number to get the scan code
            RET
```

■ 22.6 INTEL 8279 KEYBOARD AND DISPLAY CONTROLLER

Intel 8279 is a specialized I/O chip that has two major functions. It takes care of scanning a matrix keyboard, as well as refreshing a multiplexed display so that the processor is relieved of these mundane tasks. A very brief description of the working of 8279 is indicated in the following two paragraphs and details are provided later.

The 8279 scans a matrix keyboard continuously. During the scan if a key on the keyboard is found pressed, it performs the scan again after about 10 μs, allowing for the key bounce to die down. If the key is still found pressed, it is interpreted as a valid key depression. Then the scan code of the key pressed is stored in the FIFO RAM. The scan code indicates the location of the key in terms of row position and column position, as well as the status of shift and ctrl keys. FIFO RAM is inside the 8279 having a size of eight words, each of 8 bits width. Then IRQ, the interrupt request output, is activated by the 8279. The IRQ output is connected to an interrupt request pin of the microprocessor. The processor reads the FIFO RAM on a first-in first-out basis to receive from the 8279 the scan code of the key pressed. As the scanning, debouncing, and storing of scan code are all automatically performed by the 8279, the processor is relieved of these mundane tasks.

The 8279 is also used for refreshing multiplexed display consisting of as many as sixteen 7-segment LEDs. The 7-segment codes for the characters to be displayed are stored in the display RAM. Display RAM is inside the 8279 having a size of 16 words, each of 8-bits width. There are 16 locations in the display RAM corresponding to the sixteen 7-segment LEDs. The 8279 automatically sends out the 7-segment code from a display RAM location and displays a character on the corresponding LED. After a short time of about half a millisecond, the LED that is presently displaying a character is switched off. The 7-segment code from the next display RAM location is then sent

out and a character is displayed on the corresponding LED. The operation is continued until all the 16 LEDs are lighted up one at a time. This cycle takes only about 10 μs. The cycles are then repeated endlessly so that about 100 cycles are performed in a second. Because of the principle of 'persistence of vision', all the 16 LEDs seem to display the characters simultaneously and continuously. As all the aforementioned tasks are performed automatically by the 8279, the processor is relieved of these mundane tasks.

22.6.1 Pins of 8279

Intel 8279 is a 40-pin programmable IC available as a DIP package. Its physical and functional pin diagrams are indicated in Figs. 22.14 and 22.15, respectively.

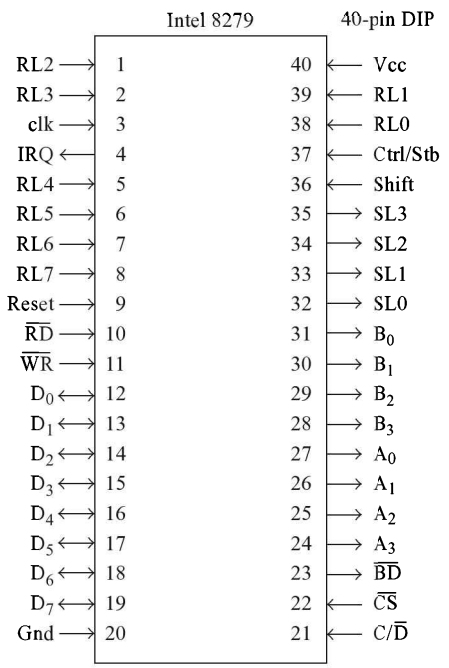


Fig. 22.14
Physical pin diagram of Intel 8279

Following is the description of the pins of 8279. The reader is advised to read this portion once again after the working of 8279 is explained.

Vcc and Gnd: Power supply and ground pins. 8279 uses +5-V power supply.

$D_{7\text{-}0}$: Eight bi-directional data pins for communication with the processor.

C/$\overline{D}$: An address input pin. A logic 1 on this pin selects control/status buffer inside the 8279. Logic 0 on this pin selects data buffer inside the 8279. From the viewpoint of the processor, these are the only two buffers inside the 8279. See Fig. 22.16 that provides the internal architecture of 8279.

RD*: Active-low input pin that is activated by the processor to read data buffer or status buffer from the 8279. The information comes to the processor from data buffer if C/$\overline{D}$ is at logic 0. The data buffer could have received the data from FIFO/sensor RAM or display RAM. The information comes to the processor from status buffer if C/$\overline{D}$ is at logic 1.

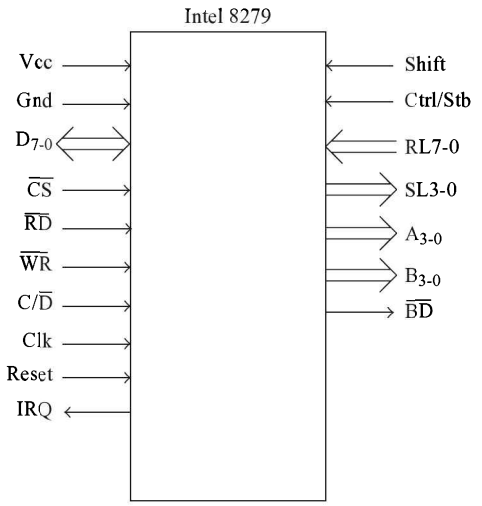


Fig. 22.15
Functional pin diagram of Intel 8279

WR*: Active-low input pin that is activated by the processor to write to data buffer or control buffer inside the 8279. The processor writes to the data buffer if C/$\overline{D}$ is at logic 0. The content of data buffer is sent to display RAM. The processor writes to the control buffer if C/$\overline{D}$ is at logic 1.

CS*: Active-low input pin used for selecting the chip.

Clk: It is the clock input pin. The maximum clock frequency on this input is 2 MHz for 8279. For 8279-5, the maximum clock frequency is 3.1 MHz. On the ALS kit, the 3-MHz clock output of 8085 is divided by 2 and the resulting 1.5-MHz frequency is provided as the clock input to 8279. This frequency is internally divided using a programmable divider by a maximum value of 31. In fact, the default value of the programmable divider is 31. This results in the internal clock frequency of about 50 kHz.

Reset: It is an active-high input pin. It is normally connected to the 'reset out' pin of 8085. As such, when the 8085 is reset, it sends out a logic 1 pulse on the 'reset out' pin thus resetting the 8279. After reset, the 8279 will be placed in the following state. The new terms that we come across are explained later.

Programmable divider value set to 31;
Display set to 16 digits in left entry mode;
Keyboard in encoded scan and two-key lockout mode.

SL_{3-0}: These are the scan line output pins. They are used to select a row to be scanned. They also select the position of the LED for display simultaneously.

If the 8279 is programmed to operate in decoded mode, the scan lines are used to drive the row lines of the matrix keyboard directly. Thus the maximum number of rows possible in the matrix keyboard would be only four. Also, there can be only four 7-segment LED digits in the display. Because of these limitations, the decoded mode is not the common mode of operation.

If the 8279 is programmed to operate in encoded mode, the LS three scan lines provide inputs to a 3 to 8 decoder. The decoder outputs are used to drive the row lines of the matrix keyboard. Thus the maximum number of rows possible in the matrix keyboard would be eight when the 8279 operates in encoded mode. Also, all the scan lines are provided as

inputs to a 4 to 16 decoder. The decoder outputs select a 7-segment LED for display. As such, there can be up to sixteen 7-segment LED digits in the display. Because of these features, the encoded mode is the common mode of operation.

However, on the ALS kit SL_3 is left unconnected. Only SL_{2-0} are used to provide inputs to a 3 to 8 decoder. The decoder output selects a row on the matrix keyboard for scanning. It also selects a 7-segment LED for display simultaneously.

RL_{7-0}: These are the eight return line input pins. The column information of a matrix keyboard or a column of information of switch sensors is received by the 8279 on these pins. These pins will be at logic 1 if no key is pressed on the row being scanned. This is because of the active internal pull-up circuitry inside the 8279. These pins also provide the 8-bit input when the 8279 is programmed to work as strobed input mode.

Shift: This input pin is connected to the Shift key on the keyboard. It is to be noted that the Shift key and Ctrl/Stb key on the keyboard do not form a part of the matrix. The information on the Shift pin is meaningful for the 8279 only if the 8279 is used for interfacing a matrix keyboard. It is not meaningful if the 8279 is used for interfacing a matrix of switch sensors. This pin is left unconnected on the ALS kit. Because of the active internal pull-up circuit, this pin will be at logic 1 on the ALS kit.

Ctrl/Stb: This input pin is connected to the Ctrl/Stb key on the keyboard. It is to be noted that the Shift key and Ctrl/Stb key on the keyboard do not form a part of the matrix. The information on the Ctrl/Stb pin is meaningful for the 8279 only if the 8279 is used for interfacing a matrix keyboard. It is not meaningful if the 8279 is used for interfacing a matrix of switch sensors. The same pin is used to receive the strobe signal when the 8279 is used as strobed input port. This pin is left unconnected on the ALS kit. Because of the active internal pull-up circuit, this pin will be at logic 1 on the ALS kit.

IRQ: It is an active-high output pin. It is normally connected to an interrupt pin of the processor. On the ALS kit, the IRQ output of the 8279 is connected to RST 5.5 interrupt request input of 8085.

If the 8279 is used for interfacing a matrix keyboard, the IRQ pin is activated when there is data in the FIFO RAM. The IRQ pin goes to logic 0 momentarily after a read operation of FIFO RAM is performed. But it returns to logic 1 if there are still data to be read from the FIFO RAM.

If the 8279 is used for interfacing a matrix of switch sensors, the IRQ pin gets activated when a change in a sensor is detected by the 8279. It is reset to 0, when the 8279 reads the sensor RAM.

B_{3-0}: These are the output pins of 8279. The LS hex digit of a display RAM location is sent out on these pins. These pins can be used to drive a 4-bit binary to 7-segment code converter. In such a case, it is possible to interface two sets of 7-segment displays, each having a maximum of 16 digits display. More commonly, these pins directly drive four of the eight segments of a 7-segment display. On the ALS kit, these pins are connected to d, c, b, and a-segments of 7-segment display unit. The correspondence among D_{3-0}, B_{3-0}, and d, c, b, a segments is as shown in the following. With this scheme, the display unit can have a maximum of 16 digits display.

D3	D2	D1	D0
B3	B2	B1	B0
d	c	b	a

A_{3-0}: These are the output pins of 8279. The MS hex digit of a display RAM location is sent out on these pins. These pins can be used to drive a 4-bit binary to 7-segment code converter.

In such a case, it is possible to interface two sets of 7-segment displays, each having a maximum of 16 digits display. More commonly, these pins directly drive four of the eight segments of a 7-segment display. On the ALS kit, these pins are connected to dp, g, f, and e-segments of the 7-segment display unit. The correspondence among $D_{7\text{-}4}$, $A_{3\text{-}0}$, and dp, g, f, e segments is as shown in the following. With this scheme, the display unit can have a maximum of 16 digits display.

D7	D6	D5	D4
A3	A2	A1	A0
dp	g	f	e

BD*: It is an active-low output status pin. This pin is activated by the 8279 when display is blanked because of a clear command or a display blank command. It is also activated during switching. This pin is left unconnected on the ALS kit.

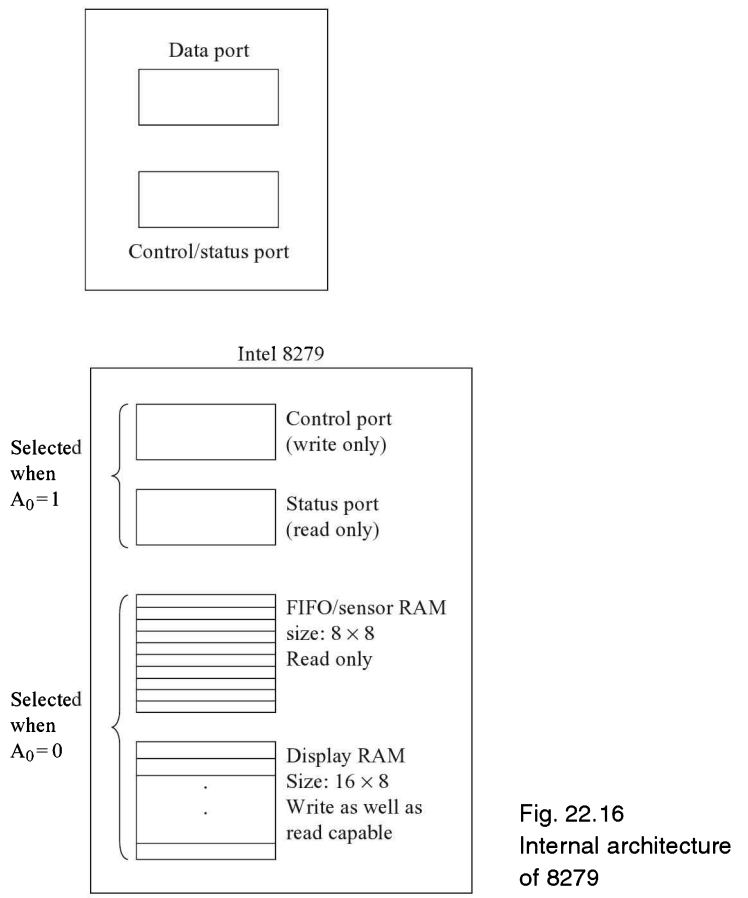


Fig. 22.16
Internal architecture of 8279

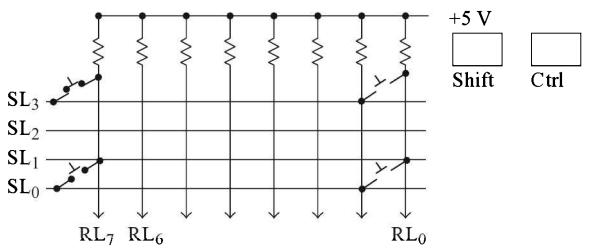
22.6.2 Keyboard Interface Considerations

The 8279 can be used to interface a matrix keyboard, like the one used in microprocessor kits or a matrix of switch sensors. It can also be used as a strobed input port.

The 'keyboard/display mode set' command (to be discussed later) indicates whether 8279 is used to interface a matrix keyboard, a matrix of switch sensors, or a strobed input port. However, very commonly the 8279 is used for interfacing a matrix keyboard rather than a matrix of switch sensors. It is very rarely, if at all, used as a strobed input port.

Interfacing matrix keyboard

Decoded mode of operation: If the 8279 is programmed to operate in decoded mode, then the matrix keyboard can have only four rows (corresponding to the four scan lines $SL_{3\text{-}0}$) and eight columns (corresponding to the eight return lines $RL_{7\text{-}0}$), for a total of $4 \times 8 = 32$ keys. At the intersection of every row and column, a key of the keyboard is connected. In addition to these 32 keys, the 'Shift' and 'Ctrl' keys could be present for a total of 34 keys. The Shift and Ctrl keys do not form a part of the matrix. Such a decoded scan matrix keyboard is shown in Fig. 22.17.



Note: Only 4 keys instead of 32 keys, are shown in the figure, for simplicity.

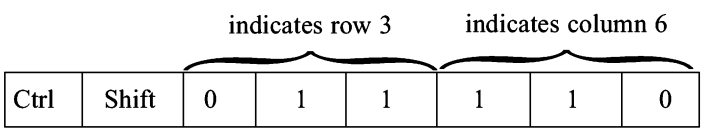
Fig. 22.17 Decoded scan matrix keyboard

The 8279 performs a scan of the matrix keyboard in decoded mode of operation as follows. One row at a time is made logic 0 and the information on the eight columns are read. The pattern sent out on $SL_{3\text{-}0}$ by the 8279 for scanning the keyboard is shown in the following table.

$SL_{3\text{-}0}$	***Row being scanned***
1110	Row 0
1101	Row 1
1011	Row 2
0111	Row 3

Because of the internal pull-up circuits, the information on a return line will be at logic 1 if the key on that column is not pressed. Hence, if all the return lines are at logic 1 it means that no key is pressed on the row scanned. If any key is pressed on the row being scanned, then logic 0 will be received on the corresponding return line by the 8279.

As an example, assume 8279 to be presently scanning row 3 by sending 0111 on $SL_{3\text{-}0}$ and the information received on the return lines $RL_{7\text{-}0}$ is 10111111, indicating that RL6 is at logic 0. Then in the FIFO RAM the following information will be stored.



The LS 3 bits indicate the column on which the key pressed is connected. Three bits are needed to provide column information, as there can be eight columns in the matrix. The LS 3 bits are 110 in the above example. It means that the key pressed is on column 6.

The next 3 bits indicate the row on which the key pressed is connected. Just 2 bits are enough to provide row information when there are only four rows in the matrix. However, 3 bits are used to provide row information, as there can be eight rows in the matrix in the encoded mode of operation. These 3 bits are 011 in the above example. It means that the key pressed is on row 3.

The Ctrl bit = 1 and Shift bit = 1 when the Ctrl and Shift keys are not pressed. In the simple matrix keyboards of the type found in microprocessor kits, the Ctrl and Shift keys are not used. Even in such cases, because of the internal pull-up circuits, these bits will be in logic 1 state.

The scan code with the pattern as described is stored in the FIFO RAM. Then the IRQ output is activated by the 8279. The IRQ output is connected to an interrupt request pin of the processor. On the ALS kit, the IRQ output is connected to RST 5.5 interrupt pin of 8085. The IRQ output becomes logic 0 when the processor reads the FIFO RAM. This is achieved by issuing 'read from FIFO RAM' command and then reading the data port of 8279.

It is possible that the processor is in disable interrupt state when the IRQ output of 8279 is activated. In such a case, the scan code of the key pressed remains in the FIFO RAM. If another key is pressed, the scan code of this key is stored in the FIFO RAM behind the scan code of the earlier key in a queue form.

Status register: There is a status register in 8279. In this the LS 3 bits, shown as NNN, provide a 3-bit field that indicates the number of characters present in the FIFO RAM. The format of the status register is shown as follows.

Du	S/E	O	U	F	N	N	N

To start with, this 3-bit field is 000. It is incremented by 1 whenever a scan code is entered into the FIFO RAM. The IRQ output is activated whenever there is data in the FIFO RAM. The IRQ output becomes logic 0 after a read operation of FIFO RAM. At this point, the 3-bit field is decremented by 1. If the 3-bit field is non-zero, the IRQ output is activated again.

If the processor remains in disable interrupt state and the user presses seven keys one after another, then the 3-bit field in the status register would change to 111. If one more key is pressed, the scan code of this key is also stored in the FIFO RAM and the FIFO RAM becomes full. This is indicated in the status register by incrementing the 3-bit field from 111 to 000 and setting the F bit to 1. Thus the LS 4 bits of status register become 1000. The F bit has the following meaning.

F = 1 means that the FIFO RAM is full.
F = 0 means that the FIFO RAM is not yet full.

If a key is pressed after the FIFO RAM is full, the scan code of the key pressed overwrites the last character in the FIFO RAM and causes over-run error. Setting of O bit in the status register indicates the over-run error.

Similarly, if the LS 4 bits of the status register are 0000, it means that the FIFO RAM is not full, and there are no characters in the FIFO RAM. In such a case, if an attempt is made to read from the FIFO RAM, it causes underflow error indicated by the setting of U flag in the status register. The meaning of the MS 2 bits of the status register is described later.

Two-key lockout mode: The scanning of the keyboard can be either two-key lockout mode or N-key rollover mode. The mode of operation is decided by keyboard/display mode set command. In the two-key lockout mode, the de-bounce logic is set when a key is pressed. Other pressed keys are looked for during the next two scans. If none are encountered, it is treated as pressing of a single key and its scan code is entered into the FIFO RAM. If another key pressing is encountered, no entry is made to the FIFO RAM.

The keyboard scanning, and de-bouncing are done for some fixed periods. It is dependent on the internal clock frequency, which is about 5 μs for keyboard scan time and 10 μs for de-bounce time when the internal clock is 100 kHz. The internal clock is derived by suitably dividing the external clock input frequency using the programmable divider present in 8279. For this purpose it is necessary to issue 'program clock' command to the 8279.

Program clock command: There are eight command words in the 8279. All of them are written to the same control port. The MS 3 bits of the control port identify the command as shown in the following table.

MS 3 bits of control port	*Command type*
000	Keyboard/display mode set
001	Program clock
010	Read FIFO/sensor RAM
011	Read from display RAM
100	Write to display RAM
101	Display write inhibit/blanking
110	Clear command
111	End interrupt/error mode set

When the MS 3 bits in the control port are 001 it means that it is program clock command. The format for the program clock command that is written to the control port of 8279 is as follows.

0	0	1	P	P	P	P	P

The LS 5 bits indicated as PPPPP provide a number in the range 2–31. Note that PPPPP should not be made 00000 or 00001. If the external clock input is 1.5 MHz, PPPPP has to be 01111 = 15 so that the internal clock frequency is 100 kHz. The default value of PPPPP after reset of 8279 is 11111 = 31.

Port addresses of the 8279 on the ALS kit: The Intel 8279 used on the ALS kit is connected in the system as an I/O-mapped I/O port with the addresses as indicated in the following.

Data port (shown as 'DAT79' in programs): D0H
Control/status port (shown as 'CTRL79' in programs): D1H

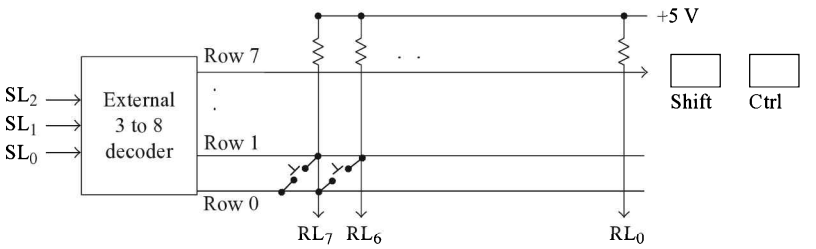
If it is desired to have the internal clock frequency as 1/15 of the external clock frequency, then the execution of the following instructions achieves this objective on the ALS kit.

```
MVI A, 00101111B
OUT D1H
```

N-key rollover mode: In the N-key rollover mode, pressing of each key is considered independent of the pressing of other keys. When a key is pressed, the de-bounce circuit waits for two keyboard scans and then checks to see if the key is still pressed. If it is, the scan code of the key is entered into the FIFO RAM. It does not mind the simultaneous pressing of other keys. If several keys are pressed simultaneously, the keys are recognized and entered into FIFO RAM according to the order the keyboard scan found them.

Encoded mode of operation: In the decoded mode of operation the maximum number of rows possible in the matrix keyboard are only four. Because of this limitation, the decoded mode is not the common mode of operation. More commonly, the 8279 is used in encoded mode of operation.

It is possible to configure the 8279 to operate in encoded scan mode by issuing the appropriate 'keyboard/display mode set' command. The matrix keyboard can then have up to eight rows and eight columns, for a total of $8 \times 8 = 64$ keys. At the intersection of every row and column, a key of the keyboard is connected. In addition to these 64 keys, Shift and Ctrl keys could be present for a total of 66 keys. The Shift and Ctrl keys do not form a part of the matrix. Such an encoded scan matrix keyboard is shown in Fig. 22.18.



Note: Only 2 keys instead of 64 keys, are shown in the figure simplicity.

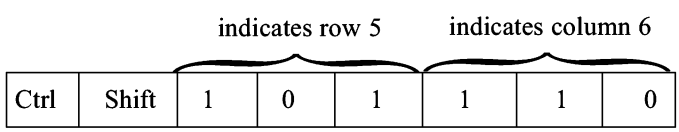
Fig. 22.18
Encoded scan matrix keyboard

As can be seen from Fig. 22.18, an external three to eight decoder is used when an encoded operation is desired. Row information is sent out on $SL_{2\text{-}0}$ in coded form and SL_3 is unused. The decoder outputs are connected to the eight rows of the matrix. Thus the operation is referred to as encoded mode. The decoder decodes the scan lines $SL_{2\text{-}0}$. The 8279 performs a scan of the matrix keyboard in encoded mode of operation as follows. One row at a time is made logic 0 and the information on the eight columns is read. The pattern sent out on $SL_{2\text{-}0}$ by the 8279 for scanning the keyboard in encoded mode is shown in the following table.

$SL_{2\text{-}0}$	***Row being scanned***
000	Row 0
001	Row 1
010	Row 2
011	Row 3
100	Row 4
101	Row 5
110	Row 6
111	Row 7

Because of the internal pull-up circuits, the information on a return line will be at logic 1 if the key on that column is not pressed. Hence, if all the return lines are at logic 1, it means that no key is pressed on the row scanned. If any key is pressed on the row being scanned, then logic 0 will be received on the corresponding return line by the 8279.

As an example, assume 8279 to be presently scanning row 5 by sending 101 on $SL_{2\text{-}0}$ and the information received on the return lines $RL_{7\text{-}0}$ is 10111111, indicating that RL_6 is at logic 0. Then in a FIFO RAM location the following information will be stored.



The LS 3 bits are 110 in the above example. It means that the key pressed is on column 6. The next 3 bits are 101 in the above example. It means that the key pressed is on row 5. The MS 2 bits provide the status of Ctrl and Shift keys. As there can be up to eight rows in the matrix keyboard when the 8279 operates in encoded mode, this mode is the preferred mode of operation in general.

Interfacing matrix of switch sensors: Intel 8279 can also be configured to scan a matrix of switch sensors such as the metal strips and magnetic sensors found on large departmental store doors and windows. In this mode, the 8279 scans all the switch sensors arranged in the form of a matrix. If it is the decoded mode of operation, the matrix size is $4 \times 8 = 32$ switch sensors, and if it is the encoded mode, the matrix size is $8 \times 8 = 64$ switch sensors. In the scanning of matrix of switch sensors, the Shift and Ctrl pins in the 8279 have no role to play. This is because, the scanning of switches, and not keys, is performed in this mode.

During the scanning of the matrix of switch sensors, the FIFO RAM works as sensor RAM. Each row of the sensor RAM is loaded with the data present on return lines during the scan of the corresponding row in the matrix of switch sensors shown as follows.

RL_7	RL_6	RL_5	RL_4	RL_3	RL_2	RL_1	RL_0

Only four sensor RAM locations are used in decoded mode of operation and all the eight sensor RAM locations are used in encoded mode of operation. If the eight-switch sensors on row 5 have the logic value as 10010011, then the return lines receive this information during the scan of row 5, and so row 5 of the sensor RAM is loaded with the value 10010011.

If a burglar tries to make a forceful entry to the departmental store by pressing a door or a window, then the switch sensor on the door or window changes its logic state. This results in the activation of IRQ output causing an interrupt to the processor. It also causes the S/E bit in the status register to be set. The S/E bit acts as S bit when the 8279 is interfaced with a matrix of switch sensors. When the S bit is set, it indicates at least one switch sensor closure. The interrupt service routine could then sound an alarm, and unchain watchdogs thus thwarting the designs of the burglars. The IRQ output is deactivated by the first data read operation of the sensor RAM if the 8279 is not in auto increment mode. Auto increment mode of operation will be described very shortly. If working in auto increment mode, the IRQ output is deactivated only by the execution of end interrupt command, which is described later. The sensor matrix will not be altered by the changes in switch sensors untill the IRQ output gets deactivated. The 8279 is rarely used for interfacing matrix of switch sensors.

Read from FIFO/sensor RAM command: To read from the FIFO/sensor RAM, the 'read from FIFO/sensor RAM' command has to be issued to the 8279 control port. The format for the read from FIFO/sensor RAM command that is written to the control port of the 8279 is as follows.

0	1	0	Ai	X	A	A	A

When the MS 3 bits in the control port are 010, it means that it is 'read from FIFO/sensor RAM' command. All the LS 5 bits in this command are don't-care bits, if the operation is reading from FIFO RAM. This is because there is no need to specify the address when reading from a FIFO RAM.

If the operation is reading from sensor RAM, then the LS 3 bits, shown as AAA, select the sensor RAM location. Three bits are needed, as there are eight locations in the sensor RAM. The bit denoted

as X is a don't-care bit. Generally a 0 is entered for this bit value. The bit denoted as Ai is the auto increment bit. The use of this bit is explained with an example.

Suppose the 8279 is interfaced with a matrix of switch sensors. Let us say 'read from sensor RAM location 3 in auto increment mode' command is issued to the command port of 8279. Then the sensor RAM pointer is loaded with address 3. Hence when the processor reads from the sensor RAM for the first time, it will be from sensor RAM location 3. Then automatically the sensor RAM pointer value is incremented to 4. So when the processor reads from the sensor RAM next time, it will be from sensor RAM location 4 automatically. Then the sensor RAM pointer value changes to 5. So when the processor reads from the sensor RAM for the third time, it will be from sensor RAM location 5. The following program segment achieves this objective on the ALS kit.

```
MVI A, 01010011B
OUT D1H   ; 'Read from Sensor RAM location 3 in Ai mode' command

IN D0H    ; Accumulator loaded with contents of row 3 of sensor RAM
MOV B, A

IN D0H    ; Accumulator loaded with contents of row 4 of sensor RAM
MOV C, A

IN D0H    ; Accumulator loaded with contents of row 5 of sensor RAM
```

Without the auto increment facility, a read from FIFO/sensor RAM command would be needed for every read operation from a sensor RAM location. Thus auto increment mode reduces programming effort.

Interfacing strobed input port: In the strobed input mode of operation the contents on the return lines of the 8279 are strobed into the FIFO RAM when the Ctrl input makes a 0 to 1 transition. The Ctrl pin is now acting like a strobe input. As such, the Ctrl pin is generally indicated as Ctrl/Stb pin. Thus the data that enters the FIFO RAM when the Ctrl/Stb makes a zero to one transition is indicated as follows.

RL_7	RL_6	RL_5	RL_4	RL_3	RL_2	RL_1	RL_0

This mode of operation is somewhat similar to the mode-1 operation of Port A or Port B of 8255 PPI chip. The 8279 is rarely used in this mode of operation.

22.6.3 Interfacing 7-Segment Display Devices

The 8279 can be simultaneously used for interfacing a matrix keyboard as well as for refreshing multiplexed display consisting of as many as sixteen 7-segment LEDs. All that is needed to display a character on a 7-segment display is just storing the corresponding 7-segment code in a display RAM location. The processor is not required to do anything else. The display RAM inside the 8279 is having a size of 16 words, each of 8-bits width.

LED connection details on the ALS kit: On the ALS kit only six 7-segment LEDs are used. The LED positions seen physically on the board and their corresponding display RAM addresses are indicated in the following. Note that there are no LEDs corresponding to display RAM locations 0 and 1. For convenience, the LEDs are numbered from 2 to 7 so that LED positions and display RAM locations correspond.

Display RAM location	7	6	5	4	3	2	1	0
LED number	7	6	5	4	3	2	–	–
Display field	Addr MS	Addr	Addr	Addr LS	Data MS	Data LS	–	–

Display RAM address	*Corresponding LED on the ALS kit*
0 to 1	No LED
2 and 3	LS and MS digit of data field
4 to 7	LS to MS digit of address field
8 to F	No LEDs

If it is required to display 'ABCD' in the address field of the ALS kit, then the following 7-segment codes have to be stored in display RAM locations 4 to 7.

Display RAM address	*Contents of display RAM*	*Seven-segment code dp g f e d c b a*	*Equivalent character*
4	A1H	1 0 1 0 0 0 0 1	D
5	C6H	1 1 0 0 0 1 1 0	C
6	83H	1 0 0 0 0 0 1 1	B
7	88H	1 0 0 0 1 0 0 0	A

Write to display RAM command: To write to the display RAM, the 'write to display RAM' command has to be issued to the 8279 control port. The format for the write to display RAM command that is written to the control port of the 8279 is as follows.

1	0	0	Ai	A	A	A	A

When the MS 3 bits in the control port are 100, it means that it is 'write to display RAM' command. The LS 4 bits, shown as AAAA, select the display RAM location. Four bits are needed, as there are 16 locations in the display RAM. The bit denoted as Ai is the auto increment bit. The use of this bit is explained with an example.

Let us say 'write to display RAM location 4 in auto increment mode' command is issued to the command port of 8279. Then the display RAM pointer is loaded with the address 4. So when the processor writes to the display RAM for the first time, it will be to display RAM location 4. Then the display RAM pointer value is automatically incremented to 5. So when the processor writes to the display RAM next time, it will be to display RAM location 5 automatically, and so on. The following program segment achieves the objective of displaying 'ABCD' in the address field on the ALS kit.

```
MVI A, 10010100B
OUT D1H      ; 'Write to Display RAM location 4 in Ai mode' command

MVI A, A1H
OUT D0H      ; Display RAM location 4 loaded with A1H

MVI A, C6H
OUT D0H      ; Display RAM location 5 loaded with C6H
```

```
MVI A, 83H
OUT D0H      ; Display RAM location 6 loaded with 83H

MVI A, 88H
OUT D0H      ; Display RAM location 7 loaded with 88H
```

At the end of the execution of the aforementioned program segment, the display RAM pointer will have the value 8.

Read from display RAM command: It is possible to read the contents of any display RAM location. To read from a display RAM location, the 'read from display RAM' command has to be issued to the 8279 control port. If the data port of 8279 is read without issuing 'read from display RAM' or 'read from FIFO/sensor RAM' command, the 8279 is not clear about the information to supply to the processor. The format for the read from display RAM command that is written to the control port of the 8279 is as follows.

0	1	1	Ai	A	A	A	A

When the MS 3 bits in the control port are 011, it means that it is 'read from display RAM' command. The LS 4 bits, shown as AAAA, select the display RAM location. Four bits are needed, as there are 16 locations in the display RAM. The bit denoted as Ai is the auto increment bit. It is to be noted that the display RAM pointer is the same for both read and write operations. The use of the Ai bit is explained with an example.

```
MVI A, 01110100B
OUT D1H      ;'Read from Display RAM location 4 in Ai mode' command

IN D0H       ;Load A with contents of Display RAM location 4
             ;After this, Display RAM pointer value will be 5.

MVI A, A1H
OUT D0H      ;Display RAM location 5 loaded with A1H
             ;After this, Display RAM pointer value will be 6.

IN D0H       ;Load A with contents of Display RAM location 6
             ;After this, Display RAM pointer value will be 7.
```

When IN D0H is executed the second time in the above example, the 8279 remembers that IN D0H refers to reading of display RAM and not reading of FIFO/sensor RAM. If our interest is to read from FIFO/sensor RAM, 'read from FIFO/sensor RAM' command has to be issued. Then subsequent read operations will be from FIFO/sensor RAM.

Refreshing of 7-segment display devices: The number of 7-segment LEDs, which can be connected using the 8279 depends on the mode of operation of the 8279.

Decoded mode of operation: In the decoded mode of operation external decoder is not made use of. One scan line at a time is made logic 0. This selects a particular LED position for the display purpose. As such, there can be a maximum of only four 7-segment LEDs. Only the first four display RAM locations are used in this mode. Hence the decoded mode of operation is rarely used. The pattern sent out on $SL_{3\text{-}0}$ by the 8279 for selecting an LED position for display is shown in the following table.

SL_{3-0}	*Selected LED*
1110	LED-0
1101	LED-1
1011	LED-2
0111	LED-3

The interfacing of common-anode 7-segment LEDs in decoded mode is shown in Fig. 22.19. First of all, the 8279 outputs 1110 on SL_{3-0}. This results in LED-0 receiving the 5-V power supply to its anode, and all other LEDs receiving 0-V power supply for their anodes.

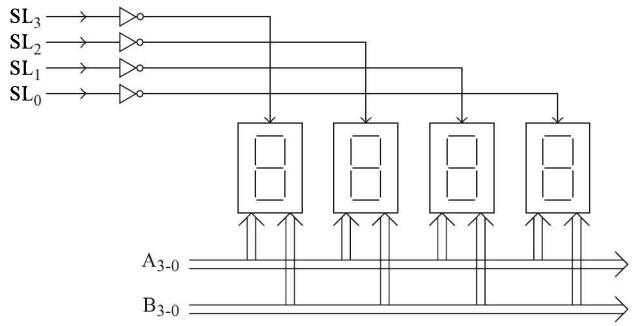


Fig. 22.19
Interfacing 7-segment LEDs in decoded mode

Meanwhile, the contents of display RAM location 0 is output on A_{3-0} and B_{3-0} by the 8279. All the 7-segment LEDs receive this information. However, a character is displayed only on LED-0 that receives the 5-V power supply.

If the internal clock frequency is 100 kHz, the clock period will be 10 μs. For 48 such clocks the signals on SL_{3-0}, A_{3-0}, and B_{3-0} will be sent. Thus the character is displayed for 480 μs duration. After 480 μs, the 8279 outputs on A_{3-0} and B_{3-0} a blanking code for 16 clock periods (160 μs) that turns off all the segments. During this period, the BD* status output is activated. The display is blanked for this duration to prevent ghosting of information from one character to the next when data on SL_{3-0} is changed to display the next character. Thus in 640 μs one character is displayed.

Next, the 8279 sends out 1101 on SL_{3-0}. This results in LED-1 receiving the 5-V power supply to its anode, and all other LEDs receiving 0-V power supply for their anodes. Meanwhile the contents of display RAM location 1 is output on A_{3-0} and B_{3-0} by the 8279. All the 7-segment LEDs receive this information. However, a character is displayed only on LED-1 that receives the 5-V power supply. This way in the next 640 μs a character on LED-1 is displayed.

Proceeding this way, all the four characters can be displayed in 4×640 μs $= 2.5$ μs. This is known as the display scan time. After this the display process is repeated endlessly to provide flicker-free display.

Encoded mode of operation: The encoded mode of operation is very commonly used as upto sixteen 7-segment display devices can be used. In the encoded mode of operation an external decoder is made use of. This mode can be configured for 8-character or 16-character display. It is configured by issuing 'keyboard/display mode set' command. If 8279 is configured for 8-character display, SL_3 is unused. Information about LED position is sent out on SL_{2-0} in coded form. An external three to eight decoder decodes the scan lines SL_{2-0}. Thus the operation is referred to as encoded mode. Only the first eight display RAM locations are used in this mode.

If 8279 is configured for 16-character display, SL_3 is also used. Information about LED position is sent out on $SL_{3\text{-}0}$ in coded form. An external 4 to 16 decoder is used in this mode. All the 16 display RAM locations are used in this mode.

The interfacing of common anode 7-segment LEDs in encoded mode is shown in Fig. 22.20. It is assumed that the 8279 is configured for a 16-character display. First of all, the 8279 outputs 0000 on $SL_{3\text{-}0}$. This results in LED-0 receiving the 5-V power supply to its anode, and all other LEDs receiving 0-V power supply for their anodes.

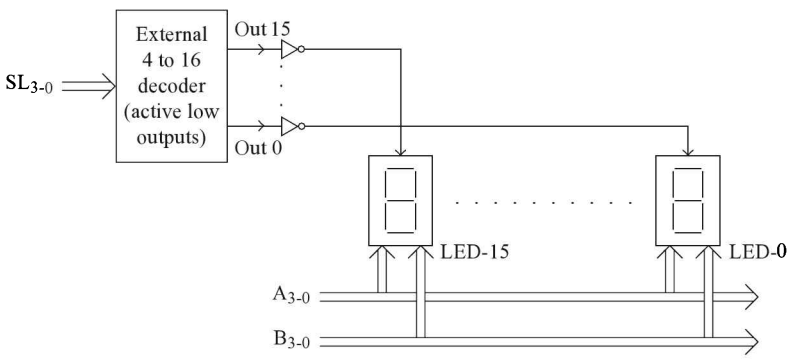


Fig. 22.20 Interfacing 7-segment LEDs in encoded mode

In the meanwhile the contents of display RAM location 0 is output on $A_{3\text{-}0}$ and $B_{3\text{-}0}$ by the 8279. All the 7-segment LEDs receive this information. However, a character is displayed only on LED-0 that receives the 5-V power supply. If the internal clock frequency is 100 kHz, as described earlier, one character is displayed in 640 μs.

Next, the 8279 sends out 0001 on $SL_{3\text{-}0}$. This results in LED-1 receiving the 5-V power supply to its anode, and all other LEDs receiving 0-V power supply for their anodes. Meanwhile, the contents of display RAM location 1 is output on $A_{3\text{-}0}$ and $B_{3\text{-}0}$ by the 8279. All the 7-segment LEDs receive this information. However, a character is displayed only on LED-1 that receives the 5-V power supply. This way a character on LED-1 is displayed in the next 640 μs.

Proceeding this way, all the 16 characters can be displayed in $16 \times 640\ \mu s = 10\ \mu s$. This is known as the display scan time. The process can be repeated 100 times per second. The display process is repeated endlessly to provide flicker-free display.

Left entry mode of display: The 8279 can be configured to operate in left entry mode or right entry mode of operation. Left entry mode is the simpler one. In this mode, the 7-segment code present at display RAM location *n* is used to display a character on LED*n*. Each 7-segment LED directly corresponds to a display RAM location.

As an example, if the 7-segment code for character 6 is entered to display RAM location 4, then character 6 is displayed on LED-4, which is the LS digit position in the address field of ALS Kit. Next, if the 7-segment code for character 7 is entered to display RAM location 5, then character 7 is displayed on LED-5. Next, if the 7-segment code for character 8 is entered to display RAM location 6, then character 8 is displayed on LED-6. Finally, if the 7-segment code for character 9 is entered to display RAM location 7, then character 9 is displayed on LED-7, which is the MS digit position in the address field. This operation is indicated in the following table. In this table '6' stands for 7-segment code for character 6, for simplicity. A blank at an LED position is indicated as b.

Entry number	*Entry value*	*Display RAM location*	*Display on LEDs* 7	6	5	4
1	'6'	4	b	b	b	6
2	'7'	5	b	b	7	6
3	'8'	6	b	8	7	6
4	'9'	7	9	8	7	6

As character codes are entered into successive display RAM locations, the characters are displayed on successive LED positions. If the LEDs were numbered from left to right, like 0, 1, ..., 7, then the display on the LEDs in the above example would be '6789'. This is very much like the appearance when characters are typed on a typewriter. Hence this mode of operation is called 'typewriter mode of entry'. On the ALS kit (like on most other kits), the LEDs are numbered from right to left. As a result the entry does not appear like typewriter entry on the ALS kit.

Right entry mode of display: The 8279 can be configured to operate in left entry mode or right entry mode of operation. Right entry mode is more complex than left entry mode. In this mode, the 7-segment code present at display RAM location *n* does not necessarily correspond to the character displayed on LED*n*.

As an example, if 8279 is configured for 8-character, right entry mode of operation and the 7-segment code for character 6 is entered to display RAM location 4, then character 6 is displayed on LED-7, which is the MS digit position in the address field. Next, if the 7-segment code for character 7 is entered to display RAM location 5, then character 7 is displayed on LED-7 and character 6 on LED-6. Next, if the 7-segment code for character 8 is entered to display RAM location 6, then character 8 is displayed on LED-7, character 7 on LED-6, and character 6 on LED-5. Finally, if the 7-segment code for character 9 is entered to display RAM location 7, then character 9 is displayed on LED-7, which is the MS digit position in the address field, character 8 on LED-6, character 7 on LED-5, and character 6 on LED-4. This operation is indicated in the following table. In this table '6' stands for 7-segment code for character 6, for simplicity. A blank at an LED position is indicated as b.

Entry number	*Entry value*	*Display RAM location*	*Display on LEDs* 7	6	5	4
1	'6'	4	6	b	b	b
2	'7'	5	7	6	b	b
3	'8'	6	8	7	6	b
4	'9'	7	9	8	7	6

As character codes are entered into successive display RAM locations, the display on all the 7-segment LEDs change. If the LEDs are numbered from left to right, like 0, 1, ..., 7, then the display on the LEDs in the above example would change in the sequence bbb6, bb67, b678, and 6789. This is similar to the appearance of the characters when typed on a calculator. Hence this mode of operation is called 'calculator mode of entry'. On the ALS kit (like on most other kits), the LEDs are numbered from right to left. As a result the entry does not appear like calculator entry on the ALS kit.

Keyboard/display mode set command: This is the command that configures the 8279 keyboard and display mode. The format for the keyboard/display mode set command that is written to the control port of 8279 is as follows.

0	0	0	D	D	K	K	K

When the MS 3 bits in the control port are 000, it means that it is 'keyboard/display mode set' command. The LS 3 bits denoted as KKK are used to configure the keyboard as indicated in the following table.

KKK	*Keyboard mode*
000	Encoded scan, 2-key lockout
001	Decoded scan, 2-key lockout
010	Encoded scan, N-key rollover
011	Decoded scan, N-key rollover
100	Encoded scan sensor matrix
101	Decoded scan sensor matrix
110	Encoded strobed input
111	Decoded strobed input

The bits denoted as DD are used to configure the display as indicated in the following table. However, if the KKK bits indicate that it is a decoded mode, the maximum number of characters in the display is only four.

DD	*Display mode*
00	8-character, left entry
01	16-character, left entry
10	8-character, right entry
11	16-character, right entry

Clear command: This command is used for clearing the display and the FIFO RAM. The format for the clear command that is written to the control port of 8279 is as follows.

1	1	0	C_D	C	C	C_F	C_A

When the MS 3 bits in the control port are 110, it means that it is clear command.

If $C_A = 1$, it means the clear command is used for 'clear all'. That is, clear both FIFO RAM and display RAM. The C_F and C_D bits are used for clearing the FIFO RAM and display RAM, respectively.

Thus, if $C_F = 1$ or $C_A = 1$, the LS 4 bits of status register are reset to 0. This means that the FIFO RAM is cleared. Similarly, if $C_D = 1$ or $C_A = 1$, the display RAM is cleared. The CC bits indicate the blanking code that is stored in all the display RAM locations, as shown in the following table.

CC	*Blanking code*
00	00H
01	00H
10	20H
11	FFH

Blanking code of 00H is generally used with common-cathode 7-segment display units. Blanking code of FFH is generally used with common-anode 7-segment display units. When the blanking code

is 20H = 00100000, all the segments except segment f will glow. This is preferred on the display of some measuring equipment, like electronic weighing machines. If the display is entirely blank, suspicion may arise about the working condition of the display.

'Du' bit of status register: The writing of the blank code in all the display RAM locations takes about 160 μs. During this period it is not possible to write to the display RAM locations. To indicate such a 'display unavailable' condition, the Du bit at the MS bit position of the status register will be set to 1 for this duration.

Display write inhibit/blanking command: The information present in a display RAM location comes out on $A_{3\text{-}0}$ and $B_{3\text{-}0}$. On $A_{3\text{-}0}$ the MS nibble is sent out, while on $B_{3\text{-}0}$ the LS nibble is sent out. Generally the 7-segment code is directly stored in a display RAM location. However, it is possible to treat $A_{3\text{-}0}$ as a hex digit and $B_{3\text{-}0}$ as another hex digit that could be displayed on two separate 7-segment displays. If the 8279 is configured for a 16-character display, it is possible to connect thirty two 7-segment display units as shown in Fig. 22.21. It makes use of two separate sets of sixteen 7-segment display units that could be called display unit A and display unit B, respectively.

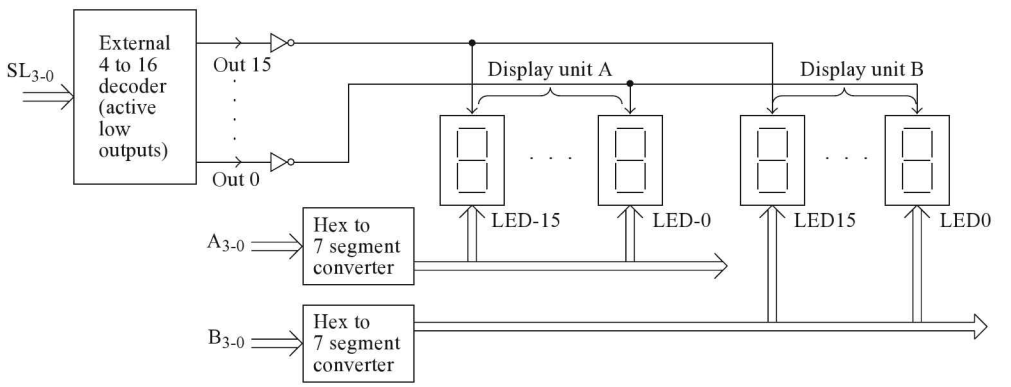


Fig. 22.21 Interfacing thirty-two 7-segment LED devices

If the 8279 is interfaced with display unit A and display unit B, sometimes it may be required to change the information displayed on only one of the display units. Sometimes it may be required to blank the display on only one of the display units. Such requirements are met using the display write inhibit/blanking command.

The format for the 'display write inhibit/blanking' command that is written to the control port of 8279 is as follows.

1	0	1	X	IW_A	IW_B	BL_A	BL_B

When the MS 3 bits in the control port are 101, it means that it is display write inhibit/blanking command. The bit shown as X in this command is a don't-care bit.

The IW bits stand for 'inhibit write'. If $IW_A = 1$ and $IW_B = 0$, the $A_{3\text{-}0}$ portion of the display RAM will not be written with any new value when a write operation to display RAM is performed. Only the

B_{3-0} portion will be altered. Similarly, if $IW_A = 0$ and $IW_B = 1$, the B_{3-0} portion of the display RAM will not be written with any new value when a write operation to display RAM is performed. Only the A_{3-0} portion will be altered.

If the contents of display RAM location 4 is C3H, it results in characters C and 3 being on the display units A and B, respectively. If it is desired to change the character 3 to character 5 on display unit B, it is achieved by making $IW_A = 1$ and $IW_B = 0$, and then writing X5H to display RAM location 4, where X is any hex digit. For example, even if the value F5H is sent to display RAM location 4, the contents change from C3 to C5, and not F5.

The BL bits stand for 'blank' display. If $BL_A = 1$, the display unit A will be blanked. The CC bits in the previously used clear command provide the code to be used for the blanking. However, the moment BL_A becomes 0, the display reappears. Thus by making $BL_A = 1$ or $BL_B = 1$, the contents of the display RAM are not changed to code for blank. If the display RAM contents are to be changed to code for blank, the clear command has to be issued to the 8279.

End interrupt/error mode set command: The format for the 'end interrupt/error mode set' command that is written to the control port of the 8279 is as follows.

1	1	1	E	X	X	X	X

When the MS 3 bits in the control port are 111, it means that it is end interrupt/error mode set command. In this command the LS 4 bits are don't care bits.

This command works as end interrupt command when the 8279 is configured for sensor matrix mode. In the end interrupt command, the E bit is also a don't care bit. When this command is executed, the IRQ output is deactivated by the 8279.

The E bit is meaningful only when the 8279 is configured for interfacing a matrix keyboard in N-key rollover mode. In such a case, if the E bit is set to 1, the 8279 will operate in special error mode. In this mode, the de-bounce cycle and key validity check are same as in normal N-key rollover mode. However, if during a single de-bounce cycle, two keys are found pressed, it will be interpreted as an error indicating simultaneous multiple key strokes. This causes S/E flag bit in the status register. The S/E bit acts like E bit (stands for error) in this case. When the S/E bit is set, it prevents further writing into the FIFO RAM and cause activation of IRQ, if not already set. The S/E bit is reset when a clear command is issued with $C_F = 1$ or $C_A = 1$.

■ 22.7 PROGRAMS USING 8279

In the previous chapters, a number of monitor routines like RDKBD, UPDAD, UPDDT, CLEAR, OUTPT were used at a number of places in the programs described. The reader need not be under the impression that writing of such monitor routines is beyond his abilities. Developing routines for these functions needs a clear knowledge of the 8279. Armed with such knowledge, routines for the above-mentioned functions are developed in this section. The names given to the equivalent routines developed in this section are shown in the following table.

ALS monitor routine	*Equivalent routine*	*Starting address*
RDKBD	KBD_RD	C6E0H
OUTPT	DISPLAY	C640H
UPDAD	ADRDISP	C6B0H
UPDDT	DATDISP	C690H
CLEAR	BLANK	C6D0H

The routines developed in this section make use of symbolic locations DATBUF and ADRBUF as indicated in the following table.

Symbolic location	*Memory address*
DATBUF (similar to CURDT)	C600H
ADRBUF (similar to CURAD)	C610H

22.7.1 KBD_RD Routine

Write an 8085 assembly language routine that returns the scan code of the key pressed on the keyboard. In other words, develop a routine that is functionally equivalent to RDKBD monitor routine.

Program

```
        ORG C6E0H
DAT79   EQU  D0H
CTRL79  EQU  D1H

KBD_RD: RIM         ; Get pending interrupt status
        ANI 00010000B; Check RST5.5 pending interrupt status
        JZ KBD_RD   ; Go to KBD_RD if RST 5.5 is not pending

        MVI A, 01000000B;
        OUT CTRL79  ; Issue command to read from FIFO RAM

        IN DAT79    ; Read scancode from FIFO RAM
        ANI 3FH     ; Reset MS two bits of scancode
        RET
```

In this routine the MS 2 bits of the scan code that correspond to Shift and Ctrl are reset to 0. This is because Shift and Ctrl keys are not present on the keyboard of the kit. The KBD_RD routine destroys the contents of only register A. This routine does not call any other routine.

22.7.2 DISPLAY Routine

Write an 8085 assembly language routine that displays two hex digits in data field, four hex digits in address field, or six hex digits on the complete six-digit display. The hex digits are stored as unpacked hex bytes in memory. Address of the first byte is provided in HL. The display field is indicated by the contents of C register shown as follows.

Contents of C register	*Display field*
00H	Data field
01H	Address field
02H	Complete field
>02H	No display

In addition, a dot is to be displayed at the end of the field if the content of B is 01H. For any other value of B register, dot is not to be displayed at the end of the field.

In other words, develop a routine that is functionally somewhat superior to OUTPT monitor routine. It is superior to OUTPT routine because OUTPT can only display in the address field or the data field. To display on all the six digits, OUTPT routine has to be called twice.

In the program that follows, DISPTBL contains a table of the 7-segment codes (without dot) for the characters '0' to '9', and 'A' to 'F'. The 7-segment codes for any character can easily be derived. For example, the derivation of 7-segment code for '3' is indicated in the following. Note that on the ALS kit, the LS bit of 7-segment is responsible for the display of segment 'a'. In other kits, the implementation could be different.

dp	g	f	e	d	c	b	a
1	0	1	1	0	0	0	0 = B0H

In the following program, the bytes 06, 05, 04, 03, 02, 01 are stored in six memory locations starting from C150H.

Memory address	*Contents*
C150H	06
C151H	05
C152H	04
C153H	03
C154H	02
C155H	01

Program

```
          ORG C640H
DISPLAY:  MOV A, C
          CPI 00H
          JZ DATA       ; Display in data field if (C) = 00

          CPI 01H
          JZ ADDR       ; Display in address field if (C) = 01

          CPI 02H
          JZ BOTH       ; Display in complete field if (C) = 02

          RET           ; Return if (C) > 2

DATA:     MVI C, 02H    ; Counter to display 2 chars in Data field
          MVI A, 92H    ; Display RAM address as 2, auto increment
          JMP DISP      ; Jump to perform display
```

```
ADDR:     MVI C, 04H   ; Counter to display 4 chars in Addr field
          MVI A, 94H   ; Display RAM address as 4, auto increment
          JMP DISP     ; Jump to perform display

BOTH:     MVI C, 06H   ; Display 6 chars in Complete field
          MVI A, 92H   ; Display RAM address as 2, auto increment

DISP:     OUT CTRL79   ; Issue 'Write to Display RAM' command
AGAIN:    MOV A, M     ; Get the character to be displayed in A
          ANI 0FH      ; Use only LS 4 bits
          XCHG         ; Save HL in DE

  ;The next 4 instructions obtain 7-segment code for the
  ;character to be displayed from the display table

          LXI H, DISPTBL;
          ADD L        ;
          MOV L, A     ;
          MOV A, M     ; Obtain 7-segment code in A
          DCR B        ;
          JNZ NO_DOT   ; Check B value for 1. If <>1, go to NO_DOT

          ANI 7FH      ; Reset MS bit of displaycode (Display dot)
NO_DOT:   OUT DAT79    ; Display char present at RAM address
          XCHG         ; Restore HL from DE

          INX H        ; Point HL to next character
          DCR C        ; Decrement counter
          JNZ AGAIN    ; Repeat till all characters displayed
          RET

DISPTBL   DB     C0H, F9H, A4H, B0H, 99H, 92H, 82H, F8H, 80H, 98H
          DB     88H, 83H, C6H, A1H, 86H, 8EH

          END
```

This routine destroys the contents of A, B, C, D, E, H, and L and does not call any other routine.

22.7.3 XPAND Routine

Write an 8085 assembly language routine that expands a two-digit hex number present in A register into two unpacked hex digits. The LS hex digit is stored in memory pointed by HL. The MS hex digit is stored in the next higher memory location.

Program

```
          ORG C6F0H
XPAND:    MOV C, A     ; Save A in C

          ANI 0FH      ; Consider only LS nibble of A
          MOV M, A     ; Save in memory pointed by HL

          MOV A, C
          RRC
          RRC
```

```
        RRC
        RRC

        ANI 0FH      ; Consider only MS nibble of A
        INX H
        MOV M, A     ; Save in next memory location
        RET
```

This routine destroys the contents of A, C, H, and L. It does not call any other routine.

22.7.4 DATDISP Routine

Write an 8085 assembly language routine that displays the contents of location DATBUF in data field. In addition, a dot is to be displayed at the end of the field if the content of B is 01H. For any other value of B register, no dot is to be displayed at the end of the field. In other words, develop a routine that is functionally the same as UPDDT monitor routine.

Program

```
        ORG C690H
XPAND   EQU  C6F0H

DATDISP: LDA DATBUF
        LXI H, DISPBUF
        CALL XPAND  ; Store LS and MS nibbles of DATBUF in
                    ; DISPBUF and DISPBUF+1

        LXI H, DISPBUF   ; HL points to characters to display
        MVI C, 00        ; Display in Data field
        CALL DISPLAY;
        RET
```

This routine destroys the contents of A, B, C, D, E, H, and L. It calls XPAND and DISPLAY routines.

22.7.5 ADRDISP Routine

Write an 8085 assembly language routine that displays the contents of location ADRBUF and ADRBUF+1 in address field. In addition, a dot is to be displayed at the end of the field if the content of B is 01H. For any other value of B register, no dot is to be displayed at the end of the field. In other words, develop a routine that is functionally the same as UPDAD monitor routine.

Program

```
        ORG C6B0H
ADRBUF  EQU  C610H
DISPBUF EQU  C620H
XPAND   EQU  C6F0H
DISPLAY EQU  C640H

ADRDISP: LDA ADRBUF
        LXI H, DISPBUF+2
        CALL XPAND  ; Store LS and MS nibbles of ADRBUF
```

```
                ;in DISPBUF+2 and DISPBUF+3

        LDA ADRBUF+1
        LXI H, DISPBUF+4
        CALL XPAND  ; Store LS and MS nibbles of ADRBUF+1
                    ;in DISPBUF+4 and DISPBUF+5

        LXI H, DISPBUF+2; HL points to characters to display
        MVI C, 01   ; Display in Address field
        CALL DISPLAY;
        RET
```

This routine destroys the contents of A, B, C, D, E, H, and L. It calls the XPAND and DISPLAY routines.

22.7.6 Program to Blank Display

Write an 8085 assembly language program that blanks the entire six-digit display.

Program

```
        ORG C000H

DAT79   EQU  D0H
CTRL79  EQU  D1H

        MVI C, 06
        MVI A, 92H
        OUT CTRL79

        MVI A, FFH  ; FFH is 7-segment code for Blank character
AGAIN:  OUT DAT79
        DCR C
        JNZ AGAIN
        HLT
```

22.7.7 BLANK Routine

Write an 8085 assembly language routine that blanks the entire six-digit display. In other words, develop a routine that is functionally the same as CLEAR monitor routine.

Program

```
        ORG C6D0H
CTRL79  EQU  D1H

BLANK:  MVI A, 11011100B ;
        OUT CTRL79 ; 'Clear command' with blank code = FFH

        RET
```

This routine destroys the contents of register A only. It does not call any routines.

22.7.8 Display Scan Code of Key

Write an 8085 assembly language program that uses the KBD_RD routine to display the scan code of the key pressed in the data field of the kit.

Program

```
          ORG C000H
UPDDT     EQU  06D3H
CURDT     EQU  FFF9H
KBD_RD    EQU  C6E0H

BEGIN:    CALL KBD_RD ; returns scan code of key pressed in A
          STA CURDT   ;
          CALL UPDDT  ; Display scan code in data field
          JMP BEGIN   ; Be in an infinite loop till Reset key is pressed
```

22.7.9 Display Characters on the Kit

Write an 8085 assembly language program that uses the DISPLAY routine to display '123456.' on the six-digit display of the kit.

Program

```
          ORG C000H
DAT79     EQU  D0H
CTRL79    EQU  D1H
DISPLAY   EQU  C640H

          LXI H, C150H     ; Characters are stored from C150H
          MVI C, 02H       ; Display in the complete field
          MVI B, 01H       ; Display a dot at the end of the field
          CALL DISPLAY     ; Display the characters
          HLT
```

In the above program, the characters to be displayed are stored in memory locations starting from C150H shown as follows.

Memory address	*Memory contents*
C150H	06
C151H	05
C152H	04
C153H	03
C154H	02
C155H	01

22.7.10 Use of DATDISP Routine in a Program

Write an 8085 assembly language program that uses DATDISP routine in a main program to display '56.' in the data field of the kit.

Program

```
          ORG C000H
DATBUF    EQU  C600H
DISPBUF   EQU  C620H
DATDISP   EQU  C690H

          MVI A, 56H    ;
          STA DATBUF    ; Store 56 in DATBUF
          MVI B, 01H    ; Display a dot at the end of the field
          CALL DATDISP  ; Display '56.' in Data field
          HLT
```

22.7.11 Use of ADRDISP Routine in a Program

Write an 8085 assembly language program that uses ADRDISP routine in a main program to display '1234.' in the address field of the kit.

Program

```
          ORG C000H
ADRBUF    EQU  C610H
DISPBUF   EQU  C620H
ADRDISP   EQU  C6B0H

          LXI H, 1234H ;
          SHLD ADRBUF   ; Store 34 in ADRBUF and 12 in ADRBUF+1
          MVI B, 01H    ; Display a dot at the end of the field
          CALL ADRDISP  ; Display '1234.' in Address field
          HLT
```

22.7.12 Use of BLANK Routine in a Program

Write an 8085 assembly language program that uses BLANK routine in a main program to blank the display on the kit.

Program

```
          ORG  C000H
BLANK     EQU  C6D0H

          CALL BLANK; Blank the display
          HLT
```

22.7.13 Program to Change the Contents of a Memory Location

Write an 8085 assembly language program that changes the contents of a RAM location on the kit without using any of the monitor routines supplied by the manufacturer of the kit. The following actions should take place when the program is executed.

1. The entire display is blanked out to start with.
2. Keep displaying the characters input from the keyboard in the address field in 'moving display fashion' till the 'Next' key on the kit is pressed.
3. Keep displaying the characters input from the keyboard in the data field in 'moving display fashion' till the 'Exec' key on the kit is pressed.
4. Contents of memory whose address is provided in the address field should be changed to the value displayed in the data field.

The program, which follows makes use of KBD_RD routine developed in this chapter. It does not make use of any of the monitor routines supplied along with the kit.

```
          ORG C010H
DAT79     EQU D0H
CTRL79    EQU D1H
EXEC      EQU 10H  ; Scancode of 'Exec' key is 10H
NEXT      EQU 11H  ; Scancode of 'Next' key is 11H
DATBUF    EQU C600H
ADRBUF    EQU C610H
KBD_RD    EQU C6E0H
          MVI A, 00H
          OUT CTRL79   ; mode set for 8-char left entry

          MVI A, DFH
          OUT CTRL79   ; Clear command with FFH as blank code

          LXI D, 0000
AGAIN:    CALL KBD_RD  ; Read from keyboard. Scan code in A
          MOV B, A     ; Save A in B

          CPI NEXT
          JZ DATFLD

; The next 19 instructions perform shift left of DE pair by a hex digit.
; In the vacancy created at the LS digit position, the new hex digit
; received from keyboard is inserted.
; For example, if DE were 1234H and 5 were entered from keyboard, DE
; changes to 2345H

          MOV A,D   ;(A)=12H
          ANI 0FH   ;(A)=02H
          MOV D,A   ;(D)=02H

          MOV A,E   ;(A)=34H
          ANI F0H   ;(A)=30H
          ORA D     ;(A)=32H

          RLC
          RLC
          RLC
          RLC       ;(A)=23H
          MOV D,A   ;(D)=23H
```

```
        MOV A,E   ;(A)=34H
        ANI 0FH   ;(A)=04H

        RLC
        RLC
        RLC
        RLC       ;(A)=40H

        ADD B     ;(A)=45H
        MOV E,A   ;(E)=45H

        XCHG
        SHLD ADRBUF
        XCHG    ; Save DE contents in ADRBUF and ADRBUF+1
        MOV A,B ; Get scan code of key pressed into A from B

        LXI H, TABLE
        ADD L
        MOV L,A
        MOV B,M ; Get 7-segment code of key pressed into B

        MVI A, 76H ; 76H = 01110110B
        OUT CTRL79 ; Read from display RAM location 6 in auto increment
        IN DAT79
        OUT DAT79 ; Read from display RAM location 6 and write to location 7

        MVI A, 75H
        OUT CTRL79
        IN DAT79
        OUT DAT79; Read from display RAM location 5 and write to location 6

        MVI A, 74H
        OUT CTRL79
        IN DAT79
        OUT DAT79; Read from display RAM location 4 and write to location 5

        MVI A, 84H
        OUT CTRL79  ; 'Write to Display RAM location 4' command

        MOV A,B
        OUT DAT79 ;Write 7-segment code of key pressed into display RAM
                  ;location 4
        JMP AGAIN

DATFLD: MVI C, 0    ;C will finally have 2-digit hex value input from keyboard
AGAIN1: CALL KBD_RD ;Input from keyboard. Scan code in A
        MOV B,A     ;Save A in B

        CPI EXEC
        JZ EXIT     ;If 'Exec' key pressed, go to EXIT
```

```
; The next 9 instructions perform shift left of C register by a hex
; digit. In the vacancy created at the LS digit position, the new hex
; digit received from keyboard is inserted. For example, if C were 34H
; and 5 were entered from keyboard, C changes to 45H
      MOV A,C ;(A)=34H
      ANI 0FH ;(A)=04H

      RLC
      RLC
      RLC
      RLC      ;(A)=40H

      ADD B    ;(A)=45H
      STA DATBUF
      MOV C,A ;(C)=45H

      MOV A,B; Get scan code of key pressed into A from B
      LXI H, TABLE
      ADD L
      MOV L,A
      MOV B,M; Get 7-segment code of key pressed into B

      MVI A, 72H
      OUT CTRL79
      IN DAT79
      OUT DAT79  ; Read from display RAM location 2 and write to location 3

      MVI A, 82H
      OUT CTRL79 ;'Write to Display RAM location 2' command

      MOV A,B
      OUT DAT79 ;Write 7-segment code of key pressed into display RAM
                ;location 2

      JMP AGAIN1

EXIT: LHLD ADRBUF; HL loaded with address present in the address field
      LDA DATBUF  ; A loaded with data present in data field
      MOV M,A

      HLT

TABLE DB  C0H, F9H, A4H, B0H, 99H, 92H, 82H, F8H
      DB  80H, 98H, 88H, 83H, C6H, A1H, 86H, 8EH

      END
```

22.7.14 Program to Alternately Display and Blank

Write an 8085 assembly language program to display 123456 on the kit and then blank the kit when a key is pressed. Again display 123456 when a key is pressed and repeat it again and again. Demonstrate the use of 'display write inhibit/blanking' command.

```
          ORG C010H
DAT79     EQU D0H
CTRL79    EQU D1H
```

```
CURAD     EQU FFF7H
CURDT     EQU FFF9H
UPDAD     EQU 06BCH
UPDDT     EQU 06D3H
RDKBD     EQU 0634H

          MVI A, 11011111B
          OUT CTRL79       ; Clear display

          MVI A, 00001110B
          SIM
          EI    ; Unmask RST 5.5 and Enable Interrupts

          CALL RDKBD       ; Wait for pressing of a key
          LXI H, 1234H
          SHLD CURAD

          MVI A, 56H
          STA CURDT

          CALL UPDAD       ; Display 1234 in Address field
          CALL UPDDT       ; Display 56 in Data field
LOOP:     CALL RDKBD       ; Wait for pressing of a key

          MVI A, 10100011B
          OUT CTRL79       ; Blank display

          CALL RDKBD       ; Wait for pressing of a key
          MVI A, 10100000B
          OUT CTRL79       ; Resume display

          JMP LOOP
          END
```

22.7.15 Program to Check for the Availability of Display

Write an 8085 assembly language program that performs the following.

1. Issue command to clear display.
2. Read status register to find out if display RAM is available for making changes.
3. If it is not available, keep storing 00H in consecutive memory locations till display RAM becomes available.
4. Store FFH in the next memory location when display RAM becomes available.

```
          ORG C010H
DAT79     EQU D0H
CTRL79    EQU D1H

          MVI A, DFH; DFH = 11011111
          OUT CTRL79; Clear display

          LXI H, C300H
AGAIN:    IN CTRL79 ; Read Status register
```

```
            RLC
            JC NOTAVAI; If MS bit = 1, Display RAM is not available

            MVI M, FFH
            HLT       ; Store FF in memory and Halt if Display available

NOTAVAI:    MVI M, 0
            INX H
            JMP AGAIN ; Store 00 in consecutive memory locations

            END
```

[redacted]

1. Write a circuit diagram to interface a 7-segment display unit using a simple latch. What are the disadvantages of this scheme?
2. Write an 8085 assembly language program that displays '34567890' continuously on the eight 7-segment LEDs, assuming that the LEDs are interfaced with the microprocessor as in Fig. 22.6.
3. Explain with a neat diagram the display interface that uses serial data transfer. Write an 8085 assembly language program that displays alternately '1234' and '5678' with a gap of 2 s between them.
4. Write a circuit diagram to interface a simple keyboard using tri-state buffers. What are the drawbacks of this method?
5. Explain with a neat diagram the matrix keyboard interface using 8255 ports. What are the disadvantages of using 8255 ports for interfacing a matrix keyboard?
6. Write an 8085 assembly language program that uses 3×8 matrix keyboard interface, and displays the scan code of the key that is pressed on the interface.
7. Provide a brief overview of the working of 8279 keyboard/display controller chip and describe its functional pin diagram.
8. Compare the decoded and encoded modes of operation of 8279 with respect to interfacing of matrix keyboards and interfacing 7-segment display devices.
9. What is FIFO/sensor RAM? Describe its use in 8279. Explain how a location of FIFO/sensor RAM can be read?
10. Describe the status register of 8279.
11. Compare two-key lockout and N-key rollover modes of operation.
12. Explain how the 8279 could be used to interface a matrix of switch sensors?
13. What is display RAM? Describe its use in 8279. Explain how a location of display RAM can be read or written?
14. Explain left entry and right entry modes of operation.

15. Explain keyboard/display mode set command of 8279.
16. Explain the following commands of 8279

    a. Clear command;
    b. Display write inhibit/blanking command;
    c. End interrupt/eror mode set command.

17. Develop your own routine that is functionally equivalent or superior to a monitor routine provided on your kit.

23

Intel 8259A—Programmable Interrupt Controller

- Need for an interrupt controller
- Overview of the working of 8259
- Pins of 8259
- Registers used in 8259
  - *Interrupt request register*
  - *Interrupt mask register*
  - *In-service register*
  - *Slave register*
- Programming the 8259 with no slaves
  - *Initialization command word 2 (ICW2)*
  - *Initialization command word 1 (ICW1)*
  - *Initialization command word 3 (ICW3)*
  - *Initialization command word 4 (ICW4)*
  - *Operation command word 1 (OCW1)*
  - *Operation command word 2 (OCW2)*
  - *Operation command word 3 (OCW3)*
- Programming the 8259 with slaves
  - *Initialization command word 3 (ICW3)*
  - *Initialization command word 4 (ICW4)*
- Use of 8259 in an 8086-based system
- Architecture of 8259
- Questions

In this chapter we deal with the above mentioned topics of programmable interrupt controller 8259A. For simplicity, the 8259A is denoted as 8259 throughout this chapter.

■ 23.1 NEED FOR AN INTERRUPT CONTROLLER

In an 8085 microprocessor, there are five interrupt input pins. They are TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR. If there are a maximum of five I/O devices that desire to perform interrupt driven data transfer, they can be connected to the five interrupt input pins. Now consider the case where there are more than five I/O devices that would like to perform interrupt driven data transfer. In such a case, on some interrupt pins more than one I/O device will have to interrupt. In fact, most microprocessors provide very few interrupt input pins. For example, Zilog Z-80, Motorola 6800, Intel 8086 have only two interrupt input pins. Thus an interrupt pin may be required to receive interrupt requests from several devices. Figure 23.1 shows a case where three I/O devices interrupt the 8085 using the RST 6.5 interrupt input.

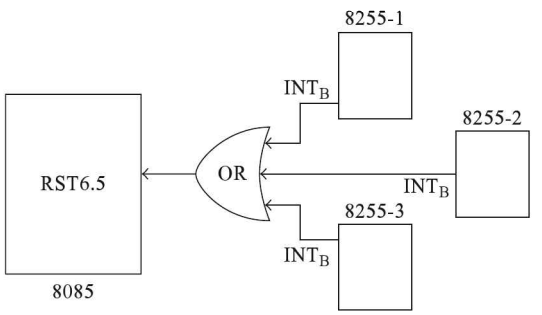


Fig. 23.1
Three devices interrupting 8085 using RST6.5 pin

In Fig. 23.1, the inputs to the OR gate are from Port B of 8255 PPI chips. Port B in these 8255 chips are assumed to be configured for mode 1 operation. The 8085 is interrupted when any one or more of the three inputs to the OR gate is active. It then branches to the interrupt service subroutine (ISS) at $6.5 \times 8 = 52 = 0034$H. Thus the branch to ISS at 0034H takes place immaterial of the source of the interrupt. In reality, the processor is required to branch to different subroutines based on the source of the interrupt. Thus in the ISS for RST 6.5 the processor has to identify from among the I/O ports connected to the RST 6.5 pin the I/O port that needs service. This process of identifying the port that needs service is called 'polling'. The subroutine that starts at 0034H is now called the interrupt level subroutine (ILS) for RST 6.5 instead of ISS for RST 6.5. Once the port that needs service is identified in the ILS, the processor performs a branch to the appropriate ISS.

The identification of the port that needs service is quite simple. The status information about Port B is obtained by reading Port C. If the LS bit of Port C is 1, it means that Port B interrupt request is active. The format of the ILS is shown as follows.

```
PUSH PSW
PUSH B
PUSH D
PUSH H      ; Save registers

IN PORT_C1
RRC
JC DEV_1    ; If LS bit of Port C in 8255-1 is 1, go to DEV_1

IN PORT_C2
RRC
```

```
        JC DEV_2     ; If LS bit of Port C in 8255-2 is 1, go to DEV_2
        IN PORT_C3
        RRC
        JC DEV_3     ; If LS bit of Port C in 8255-3 is 1, go to DEV_3
        JMP EXIT
DEV_1:CALL ISS_1     ; Branch to ISS_1
        JMP EXIT
DEV_2:CALL ISS_2     ; Branch to ISS_2
        JMP EXIT
DEV_3:CALL ISS_3     ; Branch to ISS_3
EXIT: POP H
        POP D
        POP B
        POP PSW      ; Restore registers
        EI
        RET          ; Return to interrupted program
ISS_1:-
        -
        -
        -
        RET
ISS_2:-
        -
        -
        -
        RET
ISS_3:-
        -
        -
        -
        RET
```

The disadvantages of the polling method are as follows.

1. The polling process as shown in the ILS earlier takes quite a lot of time. This results in slow interrupt response.
2. If all the three inputs of the OR gate are active in Fig. 23.1, then in the ILS shown, device_1 is serviced first as device_1 is polled first. Thus the priority for the various devices connected on a single interrupt input pin is decided by the order in which they are polled. Similarly, if only device_3 is in need of service, the ILS first of all checks device_1 and device_2, finds that they are not in need of service, and only then it checks device_3 and starts servicing it. The priority thus gets fixed and cannot be altered.

The above mentioned disadvantages are eliminated if a programmable interrupt controller (PIC) is used in the microcomputer system. The 8259 chip is such a PIC in the Intel family.

■ 23.2 OVERVIEW OF THE WORKING OF 8259

The 8259 accepts interrupt requests from as many as eight interrupting devices on IR_0 to IR_7 pins. Then it identifies the highest priority interrupt request from among those inputs that are active. It is possible to configure the 8259 for 'fixed priority' mode of operation. In such a case, IR_0 has the highest and IR_7 has the lowest priority. If IR_2, IR_4, and IR_6 inputs are active, then IR_2 is the highest priority interrupt request among the active requests. The details about the interrupt requests that are active are stored in the interrupt request register (IRR).

It is possible to mask the interrupt requests by loading the interrupt mask register (IMR). If IR_2 and IR_3 interrupt requests are masked, then IR_4 is the highest priority interrupt request among the active requests that are not masked. It is possible that the processor is already servicing IR_5 interrupt request. Information about the interrupt requests that are presently being serviced will be kept in in-service register (ISR).

There is a priority resolver unit in the 8259. It receives inputs from IRR, IMR, and ISR and identifies the highest priority interrupt request. As the priority of IR_4 is greater than IR_5 that is currently being serviced, the INT (interrupt request) output is activated. At the same time, bit 4 of ISR is set to 1 by the 8259. The INT output of 8259 is connected to INTR input of 8085 as shown in Fig. 23.2. The INT output of 8259 should not be connected to any other interrupt pin of 8085.

Thus the priority resolver decides to activate INT output only when the following conditions are satisfied.

- An IR input is activated;
- The IR input is not masked;
- The processor is presently not servicing an IR request with a higher priority.

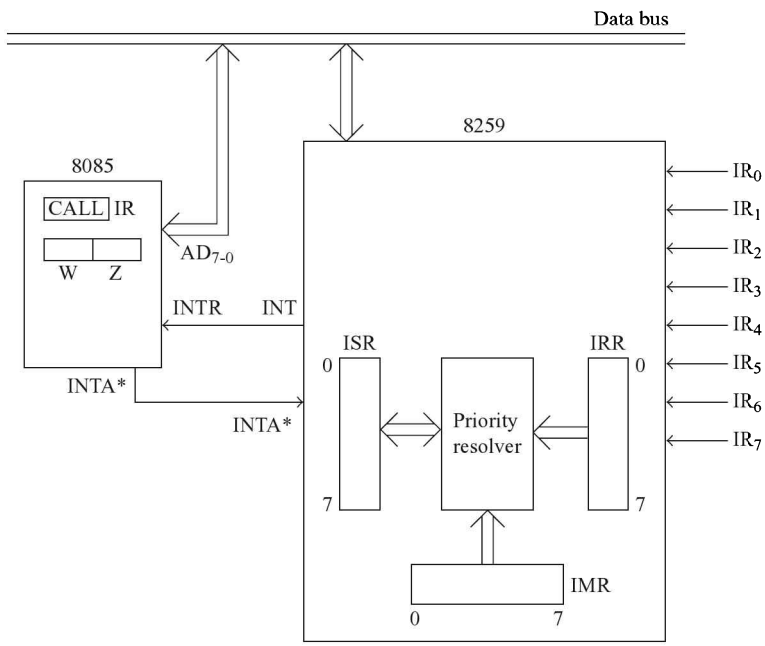


Fig. 23.2 Interfacing of 8259 with 8085 processor

The 8085 completes the execution of the instruction during which the INTR input was activated. Then the 8085 sends out INTA* output thrice in succession assuming that the 8085 interrupt system is enabled, and higher priority interrupts of 8085 are not active. In response to the activation of INTA*, the 8259 sends to the 8085 using the $D_{7\text{-}0}$ pins a 3-byte CALL instruction. The first time the INTA* is activated, the 8259 sends code for CALL (=CDH) to the 8085 on $D_{7\text{-}0}$ pins. It is received in the IR register of 8085. The second time the INTA* is activated the 8259 sends LS byte of interrupt vector (IV) address to the 8085 on $D_{7\text{-}0}$ pins. It is received in the Z register of 8085. The third time the INTA* is activated the 8259 sends MS byte of IV address to the 8085 on $D_{7\text{-}0}$ pins. It is received in the W register of 8085. The IV address supplied by the 8259 to the 8085 depends on the IR input of 8259 that is being serviced. This results in a branch to the appropriate ISS. After finishing the ISS the control returns to the main program.

The important thing to note is that the processor is not required to identify the source of the interrupt on INTR pin. The 8259 has the mechanism to identify the source of interrupt from among IR_0 to IR_7. It sends to the 8085 the CALL instruction with appropriate ISS address accordingly. Thus the problem of polling is eliminated and so the interrupt response time is reduced. Second, the 8259 could be configured to operate in 'rotating priority' mode. Then the disadvantage of fixed priority is also taken care of.

With a single 8259 in the system as many as eight interrupting devices can interrupt on INTR input of 8085. If there are a large number of devices that need to perform interrupt driven data transfer, multiple 8259s could be used. The 8259 whose INT output is connected to INTR input of 8085 will be called the Master 8259. The 8259s whose INT outputs are connected to the IR inputs of the Master 8259 will be called Slave 8259s. With this scheme a total of 64 devices can interrupt on the INTR input of 8085. Such a scheme is shown in Fig. 23.3. In this figure only two Slave 8259s are used. As

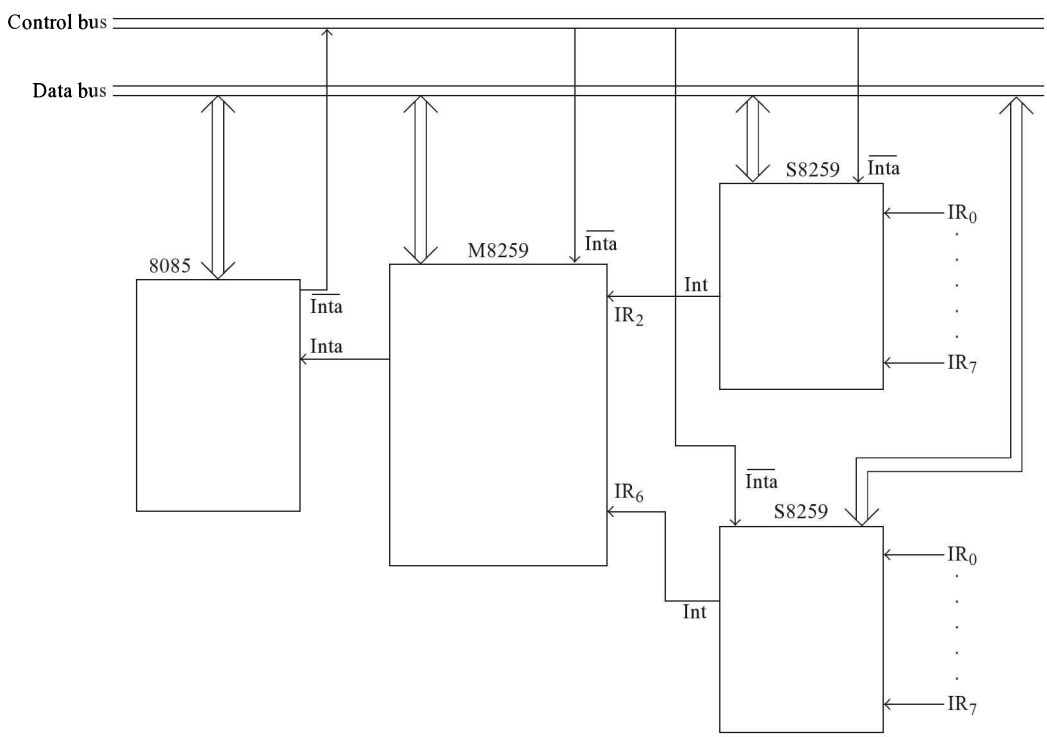


Fig. 23.3 Use of multiple 8259s

such, $2 \times 8 + 6 = 22$ devices can interrupt on the INTR input of 8085. The 8259 can be used even in Intel 8086-based system. Intel 8086 is a 16-bit processor. The working of 8259 in a 8086-based system is described at the end of the chapter.

■ 23.3 PINS OF 8259

Intel 8259 is a 28-pin programmable IC available as a DIP package. Its physical and functional pin diagrams are indicated in Figs. 23.4 and 23.5, respectively.

Vcc and Gnd: Power supply and ground pins. The chip uses +5-V power supply.

$D_{7\text{-}0}$: Eight bi-directional data pins for communication with the processor.

RD*: Active low-input pin that is activated by the processor to read status information from the 8259.

WR*: Active low-input pin that is activated by the processor to write control information to the 8259.

CS*: Active low-input pin used for selecting the chip.

A_0: An address input pin. It is used along with RD*, and WR* to identify the various command words sent to the 8259 and the status read from the 8259.

IR_0–IR_7: Eight asynchronous interrupt request inputs. The interrupt requests can be programmed for level-triggered mode or edge-triggered mode.

INT: It is an active high-output pin that interrupts the processor. It is always connected to INTR interrupt input of 8085. The INT output is activated only when all the following conditions are satisfied.

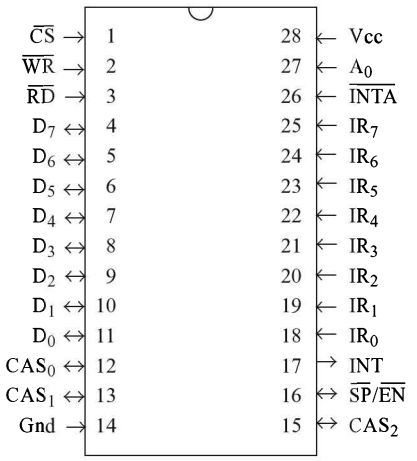


Fig. 23.4
Pin diagram of Intel 8259

- An IR input is activated.
- The IR input is not masked.
- The processor is presently not servicing an IR request with a higher priority.

INTA*: It is an active low-input pin. The 8259 receives this signal from INTA* output of 8085. When the 8085 sends three successive INTA* signals, the 8259 sends a 3-byte CALL

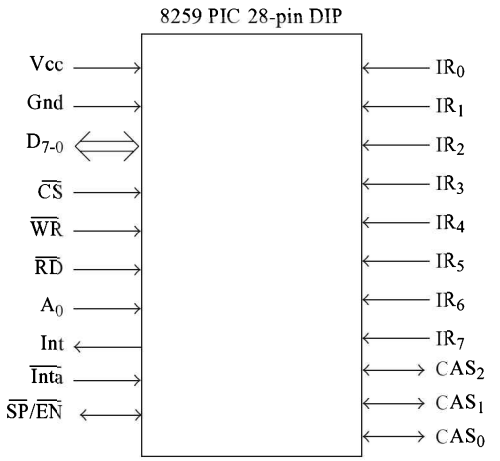


Fig. 23.5
Functional pin diagram of 8259

instruction to the 8085 using the $D_{7\text{-}0}$ pins. The second and third bytes of the CALL instruction contain ISS address that depends on the IR input of 8259 that is being serviced.

$CAS_{2\text{-}0}$: These are the cascade lines. They are used only when there are multiple 8259s in the system. The interrupt control system can have a Master 8259 and a maximum of eight Slave 8259s. The INT output of Master 8259 is connected to INTR input of 8085. The INT output of a Slave 8259 is connected to an IR input of Master 8259.

For the Master 8259, CAS pins are output pins that are used to identify one of the Slave 8259s. For a Slave 8259, CAS pins are input pins on which the Slave 8259 receives slave identification from the Master 8259. The CAS lines form a private 8259 bus. The information sent out by Master 8259 on CAS lines is received by all the Slave 8259s on their CAS inputs.

SP*/EN*: It stands for 'slave program/enable buffer'. It is a dual function pin which when used as SP* pin is an active low-input pin. If SP* = 0, the 8259 is designated as a Slave 8259. If SP* = 1, the 8259 is designated as Master 8259. When there is a single 8259 in the system, the SP* input is tied to logic 1. When used as EN* pin it is an active low-output pin that controls buffer transceivers in the buffered mode.

■ 23.4 REGISTERS USED IN 8259

From the point of view of a microprocessor, the 8259 is a specialized I/O port chip. It is never used for interfacing I/O devices, but only for controlling the interrupt system in a microcomputer. The 8259 has A_0 as the only address input pin. Thus only two addresses are possible for the 8259 ports as seen from a microprocessor. The two ports can be designated as low port and high port.

Low port is selected by the processor when $A_0 = 0$;
High port is selected by the processor when $A_0 = 1$.

The processor issues command words to these ports in order to configure the 8259 as per the need. There are several command words which are classified as initialization command words and operation command words. There are four initialization command words (ICW1, ICW2, ICW3, and

ICW4) and three operation command words (OCW1, OCW2, and OCW3). The processor also reads the status of 8259 by reading the low port and the high port. There are several status words to be read.

The 8259 makes use of a number of 8-bit registers shown as follows for its working.

Interrupt request register—IRR;
Interrupt mask register—IMR;
In-service register—ISR;
Slave register—SLR.

The processor writes command words, reads status words, or accesses registers using only the low port and the high port. Identification of a command word, status word, or a register is based on A_0 value, the bit values in certain bit positions, and in some cases by the context of the programming. The details about the identification of a command word, status word, or a register are provided later.

A brief description of the registers is provided as follows.

23.4.1 Interrupt Request Register

It is an 8-bit register that keeps track of active interrupt requests. Whenever an interrupt request input is activated, the corresponding bit in IRR register is set to 1. For example, if IR_4 and IR_6 inputs are activated, bits 4 and 6 of IRR are set to 1 making the contents of IRR as 01010000. The processor can only read the contents of this register but cannot write to IRR. To read the IRR contents, the processor has to issue OCW3 command to the 8259, with the LS 3 bits of the OCW3 command as 010. This results in 8259 storing the IRR status in low port of 8259. Then the processor has to read the low port of 8259.

23.4.2 Interrupt Mask Register

It is an 8-bit register that keeps track of the interrupt requests that are masked. If IR_4 and IR_6 requests should not cause an interrupt to the processor, it is easily achieved by making the contents of IMR as 01010000 that sets bits 4 and 6 of IMR to 1. Then even if IR_4 or IR_6 request is activated, the 8259 does not activate INT output and hence the processor will not be interrupted. The IMR is written by issuing the OCW1 command. The command uses high port of 8259. The processor can also read the contents of IMR register. To do this, the processor has to read the high port of 8259.

23.4.3 In-Service Register

It is an 8-bit register that keeps track of the interrupt requests that are currently being serviced. If IR_6 request is currently being serviced, then the contents of ISR will be 01000000. If IR_3 request becomes active during the service of IR_6, the 8259 sets bit 3 of ISR to 1 and activates INT output. But bit 6 of ISR remains set at 1 as IR_6 request is not fully serviced yet. Thus the contents of ISR will be 01001000. The following assumptions must hold good for this to happen.

- 8259 is operating in fully nested mode, without rotating priority, so that IR_3 has higher priority over IR_6.

- The processor has enabled interrupts in the routine for IR_6.
- IR_3 request has not been masked.

The processor can only read the contents of the ISR register but cannot write to ISR. To read the ISR contents, the processor has to issue OCW3 command to the 8259, with the LS 3 bits of the OCW3 command as 011. This results in the 8259 storing the ISR status in low port of 8259. Then the processor has to read the low port of 8259.

23.4.4 Slave Register

It is an 8-bit register. The processor writes to SLR but cannot read it. The content of this register has different meanings for Master 8259 and a Slave 8259. For Master 8259, it provides information about the IR inputs to which Slave 8259s are connected. If SLR of Master 8259 is loaded with the value 00001111, then it means that:

- There are Slave 8259s on IR_0, IR_1, IR_2, and IR_3.
- There are no Slave 8259s on IR_4, IR_5, IR_6, and IR_7.

For a Slave 8259, it provides information about the IR input of Master 8259 to which the Slave 8259 is connected. In this case, only the LS 3 bits of SLR are meaningful. If the SLR of a Slave 8259 is loaded with the value 00000101, then it means that the Slave 8259 is connected to IR_5 input of the Master 8259. The SLR is written by issuing the ICW3 command, which uses high port of 8259.

23.5 PROGRAMMING THE 8259 WITH NO SLAVES

In this section, it is assumed that

The processor used in the microcomputer system is 8085.
There are no Slave 8259s in the system.
No special modes of 8259 are going to be used.

Before 8259 PIC could be used in a microcomputer system for interrupt control application, it has to be properly configured to meet the needs of the system. By configuring the 8259, it is provided with a variety of information like

IV for IR_0 request;
Level-triggered or edge-triggered interrupts;
Single or multiple 8259s;
ICW4 needed or not;
Masking information for interrupt requests etc.

To provide this information to the 8259, the processor has to issue the following commands.

Initialization command word1—ICW1;
Initialization command word2—ICW2;
Initialization command word3—ICW3;
Initialization command word4—ICW4;

Operation command word1—OCW1;
Operation command word2—OCW2;
Operation command word3—OCW3.

Two initialization command words (ICW1 and ICW2) must be issued to the 8259 before it could be used, as they are compulsory. ICW3 is also compulsory provided there are Slave 8259s in the system. ICW4 is also compulsory provided the processor is 8086 or special modes of 8259 are desired. In contrast to this, operation command words are not compulsory. Hence operation command words could also be called 'optional command words'.

The command words have to be written to the low port or the high port of 8259 depending on the command.

ICW1, OCW2, and OCW3 commands are written to the low port of 8259.
ICW2, ICW3, ICW4, and OCW1 commands are written to the high port of 8259.

The 8259 port addresses depend on the chip select used in the system. For the discussion that follows, it is assumed that the port addresses of 8259 are as given in the following:

Address of low port is 50H. It is selected when $A_0 = 0$
Address of high port is 51H. It is selected when $A_0 = 1$

23.5.1 Initialization Command Word 2 (ICW2)

ICW1 will be explained after ICW2. The ICW2 command is written to the high port of 8259. It indicates $A_{15\text{-}8}$, which is the MS byte of IV address. The 8259 should supply this MS byte of IV address to the 8085 processor when the processor activates INTA* for the third time. If it is desired that the IV supplied by 8259 to 8085 processor is 2480H when IR_0 request is active, then the ICW2 command should indicate that 24H is the MS byte of IV address. This is achieved by executing the following instructions.

```
MVI A, 24H
OUT 51H
```

23.5.2 Initialization Command Word 1 (ICW1)

The ICW1 command is written to the low port of 8259. Several other commands are also written to the same port. If bit 4 of the low port is 1, the command in the low port is identified as ICW1.

The ICW1 command indicates the LS byte of IV address which the 8259 should supply to the 8085 processor when the processor activates INTA* for the second time. If it is desired that the IV supplied by 8259 to 8085 processor is 2480H when IR_0 request is active, then the ICW1 command should indicate that 80H is the LS byte of IV address.

The ICW1 command provides some additional information also to the 8259 detailed as follows.

Whether ICW4 will be issued or not.
Whether it is single or multiple 8259s in the system.
Whether interrupts are level triggered or edge triggered.
Whether the IV address interval is 4 bytes or 8 bytes.

The format of the ICW1 command is indicated as follows. Note that bit 4 = 1, indicating it is ICW1 command.

A_7	A_6	$A_{5/X}$	1	LT	4/8*	ONE	IC4

IC4: 1 = ICW4 will be issued later to the 8259.
0 = ICW4 will not be issued to the 8259.

ONE: 1 = Only one 8259 in the system.
0 = Several 8259s in the system.

4/8*: 1 = IV address interval is 4 bytes.
0 = IV address interval is 8 bytes.

LT: 1 = $IR_{0\text{-}7}$ are level-triggered interrupt requests.
0 = $IR_{0\text{-}7}$ are edge-triggered interrupt requests.

$A_{5/X}$: Is don't-care bit when 4/8* bit = 0. Provides A_5 value when 4/8* = 1.

A_7, A_6: Provide A_7 and A_6 value of IV address.

4/8 bit*: Suppose the 8259 is configured such that the IV address is 2480H when IR_0 request is activated. If IR_1 request causes the activation of INT output, the 8259 automatically sends out 2484H (a distance of 4 bytes from 2480H) as the IV address when 4/8* bit = 1. For the same IR_1 request, the IV address automatically sent out by 8259 is 2488H (a distance of 8 bytes from 2480H) when 4/8* = 0. Table 23.1 indicates the IV address sent out by 8259 for different activations IR inputs when the 4/8* bit is 1.

Table 23.1 IV addresses when configured for 4-byte interval

INT activated by	*IV address*	*LS byte of IV address*
IR_0	2480H	*100* **000** *00*
IR_1	2484H	*100* **001** *00*
IR_2	2488H	*100* **010** *00*
IR_3	248CH	*100* **011** *00*
IR_4	2490H	*100* **100** *00*
IR_5	2494H	*100* **101** *00*
IR_6	2498H	*100* **110** *00*
IR_7	249CH	*100* **111** *00*

The following points are to be noted from Table 23.1.

- Only four locations are available for writing service routine for an IR input. For example, for IR_2 request, the service routine should start at 2488H and should end by 248BH. Obviously, the amount of memory space will be inadequate. Storing a 3-byte jump instruction say 'JMP C100H' at 2488H could solve this problem. In such a case, the service routine for IR_2 starts at C100H. If the service routine for IR_2 ends at C3FFH, then the service routine for IR_3 can be started at C400H. Storing 'JMP C400H' at location 248CH achieves this.
- The LS 2 bits (shown underlined) in the LS byte of address are always 00. So the 8259 will not be informed about the value of these bits explicitly.
- The value of the next 3 bits (shown in bold type) is dependent on the IR input that is active. For IR_0 it is 000, for IR_5 it is 101 etc. The 8085 processor cannot foresee which IR request will be activated. Only the 8259 should decide about the value of these bits depending on the IR request that is active. As such, the value of these bits will not be supplied to 8259 by the 8085. Thus, out of the 16-bit IV address, the 8085 is required to supply only the MS 11 bit address. The LS 5 bits of the address will be decided by the 8259 itself. Out of the MS 11 bits of address, the MS 8 bits

of address is provided by ICW2 command, as already described. The remaining 3 bits (shown in italics) that are the MS 3 bits of LS byte of address is provided by the ICW1 command. In the ICW1 format, these 3 bits of address are indicated as A_7, A_6, and A_5.

The LS 5 bits of IV address have to be 00000 for IR_0 request. Hence the only possible IV addresses for IR_0 when 24H is the MS byte of address depends on A_7, A_6, and A_5. They are as shown in Table 23.2.

Table 23.2 Possible IV addresses for IR_0

A_7	A_6	A_5	*IV address for IR_0*
0	0	0	2400H
0	0	1	2420H
0	1	0	2440H
0	1	1	2460H
1	0	0	2480H
1	0	1	24A0H
1	1	0	24C0H
1	1	1	24E0H

Table 23.3 indicates the IV address sent out by 8259 for different activation of IR inputs when the 4/8* bit is 0, assuming IV address for IR_0 to be 2480H.

Table 23.3 IV addresses when configured for 8-byte interval

INT activated by	*IV address*	*LS byte of IV address*
IR_0	2480H	*10* **000** 000
IR_1	2488H	*10* **001** 000
IR_2	2490H	*10* **010** 000
IR_3	2498H	*10* **011** 000
IR_4	24A0H	*10* **100** 000
IR_5	24A8H	*10* **101** 000
IR_6	24B0H	*10* **110** 000
IR_7	24B8H	*10* **111** 000

The following points are to be noted from Table 23.3.

- Only eight locations are available for writing service routine for an IR input. For example, for IR_2 request, the service routine should start at 2490H and end by 2497H. Obviously, the amount of memory space will be inadequate. Storing a 3-byte jump instruction say 'JMP C100H' at 2490H could solve this problem. In such a case, the service routine for IR_2 starts at C100H. If the service routine for IR_2 ends at C3FFH, then the service routine for IR_3 can be started at C400H. Storing 'JMP C400H' at location 2498H achieves this.
- The LS 3 bits (shown underlined) in the LS byte of address are always 000. So the 8259 will not be informed about the value of these bits explicitly.
- The value of next 3 bits (shown in bold type) is dependent on the IR input that is active. For IR_0 it is 000, for IR_5 it is 101 etc. The 8085 processor cannot foresee which IR request will be activated. Only the 8259 should decide about the value of these bits depending on the IR request that is active. As such, the value of these bits will not be supplied to 8259 by the 8085. Thus, out of the 16-bit IV address, the 8085 is required to supply only the MS 10-bit address. The LS 6 bits of the address

will be decided by the 8259 itself. Out of the MS 10 bits of address, the MS 8 bits of address is provided by ICW2 command, as already described. The remaining 2 bits (shown in italics) that are the MS 2 bits of LS byte of address is provided by the ICW1 command. In the ICW1 format, these 2 bits of address are indicated as A_7 and A_6. The bit indicated as $A_{5/X}$ is used as X (don't-care) bit when 4/8* bit = 0. Generally a 0 is stored for the X bit.

The LS 6 bits of IV address have to be 000000 for IR_0 request. Hence the only possible IV addresses for IR_0 when 24H is the MS byte of address depends on A_7 and A_6. They are as shown in Table 23.4.

Table 23.4 Possible IV addresses for IR_0

A_7	A_6	*IV address for IR_0*
0	0	2400H
0	1	2440H
1	0	2480H
1	1	24C0H

Example
Assume it is required to configure the 8259 using ICW1 for the following operation.

- ICW4 is not going to be issued.
- Only one 8259 to be used in the system.
- The address interval is required to be 8 bytes.
- The $IR_{0\text{-}7}$ requests are to be edge triggered.
- The LS byte of IV address for IR_0 to be 80H.

This is achieved by executing the following instructions.

```
MVI A, 10010010B
OUT 51H
```

The MS 2 bits are 10 which means that $A_{7\text{-}6}$ = 10. Thus the LS byte of IV address for IR_0 will be 10 000000 = 80H as required.

Bit 5 is a don't-care bit. It is shown as 0 in the example shown.
Bit 4 is a 1. It indicates that the command in the high port is ICW1.
Bit 3 is 0, indicating that the 8259 is configured for edge-riggered interrupts.
Bit 2 is 0, indicating that the 8259 is configured for 8-byte interval.
Bit 1 is 1, indicating that there is only one 8259 in the system.
Bit 0 is 0, indicating that there will not be ICW4.

23.5.3 Initialization Command Word 3 (ICW3)

ICW3 is not needed when there are no Slave 8259s in the system. As it is assumed that there are no Slave 8259s in the system, the discussion about ICW3 is deferred for the present.

23.5.4 Initialization Command Word 4 (ICW4)

The ICW4 command is written to the high port of 8259 and is needed only if any of the following conditions is satisfied.

- If the processor used in the system is 8086.
- If the 8259 has to be configured for special modes (special fully nested mode, buffered mode, automatic end of interrupt mode) immaterial of the processor used in the system.

If any of these conditions is true, the processor has to issue an ICW1 command with IC4 bit, present at the LS bit position, as 1. The LS bit of ICW1 command indicates whether ICW4 command is needed or not. ICW4 command is written to the high port. Several other commands are also written to the same port. For example ICW2, as described earlier, is written to the high port of 8259. Even ICW3 and OCW1 commands are written to the same port. The 8259 identifies the command in the high port based on the ICW1 command issued to the low port of 8259 earlier.

The first time the high port is written, the command is always interpreted as ICW2. The second time the high port is written, the command is interpreted as ICW3, if there are Slave 8259s in the system. If there are no Slave 8259s, it will be interpreted as ICW4, if ICW4 is needed as indicated in bit 0 of ICW1 command. If ICW4 is also not needed, it will be interpreted as OCW1. The third time the high port is written, the command is interpreted as ICW4 if ICW4 is needed, else it is interpreted as OCW1. If the high port is written for the fourth time and onwards, the command is interpreted as OCW1. Thus if it is needed to load the high port of 8259 with ICW2 command again, it is necessary to load the low port of 8259 with ICW1 command first.

Assuming that the processor writes to the high port of 8259 five times, the interpretation of the contents of the high port will be as indicated in Table 23.5.

Table 23.5 Interpretation of the contents of high Port

When ICW3 and ICW4 are needed	*When only ICW3 is needed*	*When only ICW4 is needed*	*When ICW3 and ICW4 are not needed*
ICW2	ICW2	ICW2	ICW2
ICW3	ICW3	ICW4	OCW1
ICW4	OCW1	OCW1	OCW1
OCW1	OCW1	OCW1	OCW1
OCW1	OCW1	OCW1	OCW1

The format of the ICW4 command is indicated as follows. Note that only 5 bits are used in this command. The MS 3 bits are always 000.

0	0	0	SFNM	BUF	M/S*	AEOI	86/85*

86/85*: 1 = 8086 is the processor in the system.
0 = 8085/8080 is the processor in the system.

AEOI: 1 = Automatic end of interrupt mode (AEOI).
0 = Non-automatic end of interrupt mode (EOI).

M/S*: 1 = 8259 is configured to function as master.
0 = 8259 is configured to function as slave.

BUF: 1 = 8259 used in buffered mode.
0 = 8259 used in non-buffered mode.

SFNM: 1 = 8259 configured for special fully nested mode (SFNM).
0 = 8259 is not configured for SFNM. It is only fully nested mode (FNM).

AEOI bit: If the processor is servicing IR_4, then bit 4 of ISR is set to 1 by the 8259. This bit has to be reset to 0 sometime later. Otherwise, IR_4 input cannot interrupt the processor anymore. The highest priority ISR bit that is set can be reset to 0 by the following two methods.

- By issuing an EOI command before returning from service routine.
- Automatically following the last INTA* pulse.

If the AEOI bit is 1, the 8259 is configured for AEOI mode, else it is configured for EOI mode.

The SFNM, BUF, M/S* bits of ICW4 command will be described when 'programming the 8259 when there are slaves' topic is discussed.

23.5.5 Operation Command Word 1 (OCW1)

The OCW1 command is written to the high port of 8259. It indicates the IR inputs that must be masked. Basically, when the high port of 8259 is loaded with OCW1 command, the content of the high port is internally moved to the IMR. The format of the OCW1 command is indicated as follows.

M7	M6	M5	M4	M3	M2	M1	M0

M4: 1 = IR_4 input is masked.
0 = IR_4 input is not masked.

Similar is the meaning for the other bits in the command.

Assume IR_4 and IR_6 requests should not cause an interrupt to the processor. This is easily achieved by issuing OCW1 command with 01010000 as the data, which in turn is achieved by executing the following instructions.

```
MVI A, 50H
OUT 51H
```

23.5.6 Operation Command Word 2 (OCW2)

The OCW2 command is written to the low port of 8259. The ICW1 and OCW3 commands are also written to the same port. If bit 4 and bit 3 of the low port are 00, the command in the low port is identified as OCW2, which is used for issuing the following types of commands to the 8259.

Specific rotation, with EOI command;
Specific rotation, without EOI command;
Non-specific rotation, with EOI command;

Specific EOI, without rotation command;
Non-specific EOI, without rotation command;
Automatic EOI, with no rotation command;
Automatic EOI, with rotation command.

The format of the OCW2 command is indicated as follows. Note that bit 4 = 0 and bit 3 = 0, indicating that it is an OCW2 command.

R	SL	EOI	0	0	L_2	L_1	L_0

R: 1 = Perform rotation of priorities.
0 = No rotation of priorities.

EOI: 1 = Issue explicit EOI command.
0 = Not explicit EOI command. Generally considered as automatic EOI.

SL: 1 = Use specific level for rotate/EOI command.
0 = No specific level for rotate/EOI command.

$L_{2\text{-}0}$: Indicates the level to be used in specific rotate/EOI command. These bits are don't-care bits when SL bit is 0.

R	*SL*	*EOI*	*Configuration of 8259*
0	0	1	Non-specific EOI, no rotation
0	1	1	Specific EOI, no rotation
1	0	1	Non-specific EOI, with rotation
1	1	1	Specific EOI, with rotation
0	1	0	No operation
1	1	0	Simply rotate priorities
1	0	0	Perform rotation after AEOI
0	0	0	No rotation after AEOI

The R, SL, and EOI bits of OCW2 command identify the configuration of 8259.
When R = 1, SL = 1, and EOI = 0 in the OCW2 command, it means that specific rotation has to be performed in AEOI command. It simply rotates the priorities.

It may be noted that when R = 0, SL = 1, and EOI = 0 in the OCW2 command, it means that—no rotation, AEOI, but specified level to be used. It is indeed a 'no operation' command that is never used.

Need for rotation of priorities: In some applications it is possible that all the devices that interrupt on $IR_{0\text{-}7}$ have equal priority. However, after initialization of 8259, IR_0 will have the highest and IR_7 the lowest priority unless rotating priority is forced on 8259. This will result in a poor response to low-priority interrupt requests. For example, IR_7 request may never be serviced, if other higher priority requests keep occurring. This problem is solved using rotating priority. There are two types of rotation of priorities. They are:

Non-specific rotation;
Specific rotation.

Non-specific rotation: In this mode, the priorities will be rotated such that the highest priority ISR bit that is presently in the set state will get the lowest priority. If bit 4 of ISR is the highest-priority

ISR bit that is in the set state, then the priorities would be rotated so that IR_4 will have the lowest priority and IR_5 the highest as indicated in the following.

Priority
IR_5
IR_6
IR_7
IR_0
IR_1
IR_2
IR_3
IR_4

The EOI command is always combined with non-specific rotation command. Hence in this example, bit 4 of ISR is reset to 0. Non-specific rotation with EOI is used, when fully nested structure is preserved.

Specific rotation: In this mode, the priorities will be rotated such that the specified level will get the lowest priority. If level 4 were specified using specific rotation, then the priorities would be rotated so that IR_4 will have the lowest priority and IR_5 the highest priority as indicated in the following.

Priority
IR_5
IR_6
IR_7
IR_0
IR_1
IR_2
IR_3
IR_4

The EOI command is generally combined with specific rotation. If EOI command is combined with specific rotation in the above example shown, bit 4 of ISR will be reset to 0. Specific rotation without EOI command is not commonly used. Such a command disturbs the fully nested structure. 'Specific EOI command' will have to be executed to reset the appropriate ISR bit in case 'specific rotation without EOI command' is used earlier. An example illustrates this situation.

Consider the following situation. Presently IR_0 has the highest and IR_7 has the lowest priority. IR_4 input is activated and has not been masked. The processor is busy servicing IR_6 request. Then bit 4 of ISR will be set to 1. However, bit 6 of ISR remains set at 1, as IR_6 request has not been fully serviced. The processor now branches to the IR_4 service routine. Assume that the following instructions are executed within the IR_4 service routine.

```
MVI A, 100 00 000B
OUT 50H
```

After the execution of the instructions shown, bit 4 and bit 6 of ISR remain set, as EOI bit is 0. However priorities would be rotated so that IR_4 will have the lowest priority and IR_5 the highest priority indicated as follows.

Priority	*ISR status*
IR_5	0
IR_6	1
IR_7	0
IR_0	0
IR_1	0
IR_2	0
IR_3	0
IR_4	1

If non-specific EOI command is issued just before the end of the service routine for IR_4, the 8259 resets the highest priority ISR bit that is in the set state. In this case, it resets bit 6 of ISR and not bit 4 of ISR. The rotation of priorities without EOI command has disturbed the fully nested structure. It is necessary in such a case to issue a specific EOI command that resets bit 4 of the ISR.

Some examples for OCW2 commands are provided in the following. It is assumed that the low port address is 50H.

Example 1: The OCW2 needed to issue a non-specific EOI command to the 8259 is:

```
MVI A, 001 00 000B
OUT 50H
```

Example 2: The OCW2 needed to issue a specific EOI command to the 8259 that resets bit 5 of ISR is:

```
MVI A, 011 00 101B
OUT 50H
```

Example 3: The OCW2 needed to issue a rotate with non-specific EOI command to the 8259 is:

```
MVI A, 101 00 000B
OUT 50H
```

Example 4: The OCW2 needed to issue a rotate with specific EOI command to the 8259 that resets bit-5 of ISR is:

```
MVI A, 111 00 101B
OUT 50H
```

Example 5: If 8259 is already configured for AEOI mode using ICW4 command, then the OCW2 needed to issue a rotate in AEOI command to the 8259 is:

```
MVI A, 100 00 000B
OUT 50H
```

Example 6: If 8259 is already configured for AEOI mode using ICW4 command and the 8259 is already configured for rotate in AEOI mode using OCW2 command, as in the previous example, then the new OCW2 needed to issue a 'stop rotate in automatic EOI command' to the 8259 is:

```
MVI A, 000 00 000B
OUT 50H
```

23.5.7 Operation Command Word 3 (OCW3)

The OCW3 command is written to the low port of 8259. The ICW1 and OCW2 commands are also written to the same port. If bit 4 and bit 3 of the low port are 01, the command in the low port is identified as OCW3. The OCW3 command is used for issuing the following types of commands to the 8259.

Polled mode of operation;
Set up 8259 for reading IRR or ISR register;
Enable/disable special mask mode.

The format of the OCW3 command is indicated as follows. Note that bit 4 = 0 and bit 3 = 1, indicating it is OCW3 command. The MS bit in this command is a don't-care bit.

X	ESMM	SMM	0	1	P	RR	IS/IR*

Polled mode: When the P bit is set to 1, the 8259 works in polled mode. If P = 0, polling mode is not used. When working in polled mode, the INT output will never be activated by the 8259. As such, the INT output of 8259 need not be connected to the INTR input of 8085. In this mode, the processor has to identify the IR input that needs service using software. The processor first of all issues an OCW3 command and configures the 8259 for polled mode. Then the 8259 automatically freezes the contents of ISR based on IR inputs at that moment, till the low port of 8259 is read. Thus even if the IR inputs were to change after poll command is issued, the ISR contents do not change. The processor then reads the low port that provides information about the highest priority-level requesting service. The information is provided in the following format.

I	X	X	X	X	W_2	W_1	W_0

Where $W_{2\text{-}0}$ indicates the highest priority-level requesting service and I indicates whether the corresponding IR input is still active.

The following instructions are to be executed by the 8259 to find out about the IR input that needs service in polling mode.

MVI A, 000 01 1 00B
OUT 50H; Issue polling instruction to 8259 using OCW3 command
IN 50H; If (A) = 1 0000 101, it means IR_5 needs service and is active

The polled mode can be used to expand the number of priority levels indefinitely, unlike the limitation of 64 levels in non-polled mode using one Master and eight Slave 8259s. Figure 23.6 shows three 8259s connected in cascade. The INT output of 8259-2 is connected to IR_4 input of 8259-1 and that of 8259-3 is connected to IR_6 input of 8259-2.

Assume only IR_0 input of 8259-3 is active. All the three 8259s are configured for polled mode. The processor polls 8259-1 and comes to the conclusion that IR_4 input of 8259-1 is active. As 8259-2 is connected to IR_4 input of 8259-1, the processor polls 8259-2 and comes to the conclusion that the IR_6 input of 8259-2 is active. Finally, as 8259-3 is connected to IR_6 input of 8259-2, the processor polls

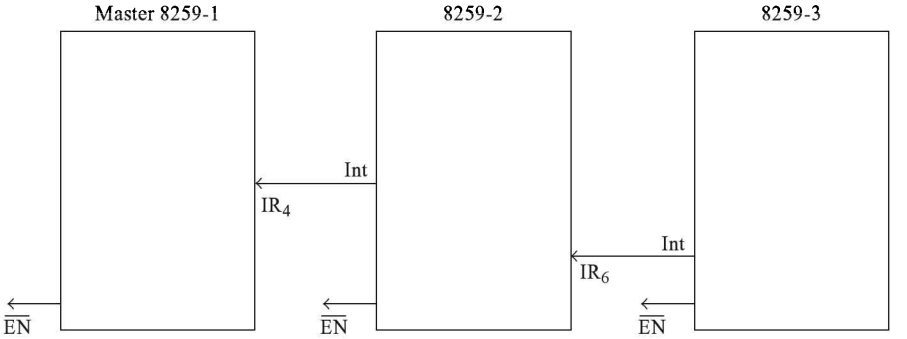


Fig. 23.6 Cascade of 8259s in polled mode

8259-3 and comes to the conclusion that the IR_0 input of 8259-3 is active. Once the interrupting source is identified, the processor branches to the appropriate service routine.

Read status of IRR or ISR: Sometimes we wish to know the contents of the IRR or ISR. For this purpose, the OCW3 command has to be executed first indicating whether it is desired to read IRR or ISR. After this, the low port of 8259 has to be read to get the status of the desired register.

RR: 1 = Configure 8259 for register read operation
0 = Not register read operation
IS/IR*: 1 = Configure 8259 for reading ISR
0 = Configure 8259 for reading IRR

The IS/IR* bit is a don't-care bit when RR bit is 0.

Thus, the execution of the following instructions results in register A of 8085 getting the status information present in IRR of 8259.

```
MVI A, 000 01 0 10B
OUT 50H; Configure 8259 for reading from IRR by issuing OCW3 command
IN 50H; Read IRR by reading the low port of 8259
```

In case OCW3 command is issued with P = 1 and RR = 1, reading of the low port provides polled information and not status of IRR or ISR.

Special masked mode (SMM): Sometimes it is necessary that in a service routine some of the lower priority IR inputs should be capable of interrupting the processor. Under normal conditions, assuming IR_0 is the highest priority input, if the processor is servicing say IR_4 request, $IR_{5\text{-}7}$ inputs cannot activate INT output of 8259. This is true even if none of $IR_{5\text{-}7}$ is masked. This is where the SMM proves useful.

If IMR is loaded with 00010011 when the 8259 is configured for SMM in the service routine for IR_4, then it means that not only the higher priority inputs IR_2 and IR_3 but also the lower-priority inputs IR_5, IR_6, and IR_7 are capable of interrupting the IR_4 service routine. For the same value in IMR, if the command were normal mask mode, only the higher priority inputs IR_2 and IR_3 would be capable of interrupting the IR_4 service routine.

The SMM is set when ESMM = 1 and SMM = 1 in OCW3 command. It reverts to normal mask mode when ESMM = 1 and SMM = 0. ESMM stands for enable special mask mode and SMM stands for special mask mode. When ESMM = 0, the SMM bit is a don't-care bit.

■ 23.6 PROGRAMMING THE 8259 WITH SLAVES

In this section, it is assumed that a Slave 8259 is connected to IR_4 input of Master 8259 and the processor used in the microcomputer system is 8085. This is shown in Fig. 23.7.

The port addresses of 8259 depend on the chip select circuit used. For the discussion, the port addresses are assumed as follows.

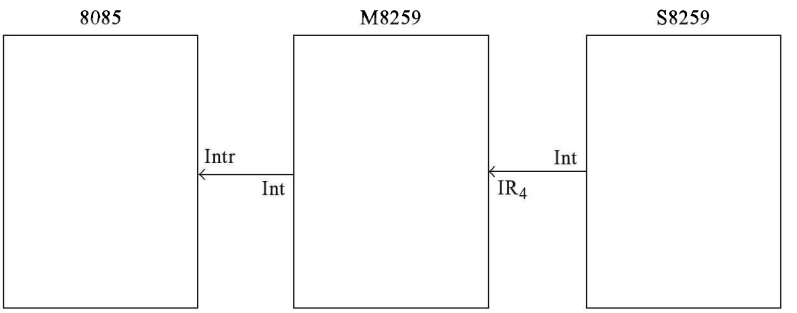


Fig. 23.7
A Slave 8259 connected to IR_4 input of Master 8259

Port	*Master 8259*	*Slave 8259*
Low port	50H	60H
High port	51H	61H

The Master 8259 and all Slave 8259s used in the system have to be individually programmed using ICWs and OCWs. The ICW1 and ICW2 have the same meaning whether it is Master 8259 or a Slave 8259. Hence discussion about ICW1 and ICW2 is not repeated.

23.6.1 Initialization Command Word 3 (ICW3)

The ICW3 command is written to the high port of 8259 and is needed only if there are Slave 8259s in the system. Bit 1 of ICW1 command indicates whether there are Slave 8259s or not. Several other commands are also written to the same port, for example, ICW2, ICW4, and OCW1. The 8259 identifies the command in the high port based on the ICW1 command issued to the low port of 8259 earlier. This topic has already been dealt with.

The ICW3 command has different interpretations for Master 8259 and a Slave 8259. Basically, when the high port of 8259 is loaded with ICW3 command, the content of the high port is internally moved to the SLR.

For Master 8259, ICW3 provides information about the IR inputs to which Slave 8259s are connected. Assume the following instructions to be executed after the ICW2 command, when there are Slave 8259s in the system.

```
MVI A, 00010000B
OUT 51H
```

The SLR of Master 8259 is loaded with the value 00010000. Then it means that there is a Slave 8259 on IR_4 input only of the Master 8259.

For a Slave 8259, ICW3 provides information about the IR input of Master 8259 to which the Slave 8259 is connected. In this case, only the LS 3 bits of SLR are meaningful. Assume SLR of Slave 8259 to be loaded with the value 00000100 using the following instructions.

```
MVI A, 0000100B
OUT 61H
```

Then it means that the Slave 8259 is connected to IR_4 input of the Master 8259.

23.6.2 Initialization Command Word 4 (ICW4)

The ICW4 was partly described earlier. It is written to the high port of 8259 and is needed only if any of the following conditions is satisfied.

- If the processor used in the system is 8086.
- If the 8259 has to be configured for special modes (special fully nested mode, buffered mode, automatic end of interrupt mode) immaterial of the processor used in the system.

If any of these conditions are true, the processor has to issue an ICW1 command with IC4 bit, present at the LS bit position, as 1. The LS bit of ICW1 command indicates whether ICW4 command is needed or not.

The format of the ICW4 command is indicated as follows. Note that only 5 bits are used in this command. The MS 3 bits are always 000.

0	0	0	SFNM	BUF	M/S*	AEOI	86/85*

The meaning of 86/85* and AEOI bits are already discussed.

Buffered mode of 8259: If the 8259 is used in a large system, then bus-driving buffers are needed on the data bus. This opens up the problem of enabling the buffers which is solved by configuring the 8259 for buffered mode of operation.

In non-buffered mode, SP*/EN* pin is used as the input pin SP*. This has been discussed earlier. However, in the buffered mode, SP*/EN* pin is used as the output pin EN* that enables the buffers. This output pin is activated whenever the 8259 data bus outputs are enabled. Then in the buffered mode, as SP*/EN* is used as EN* output, the identification of an 8259 as master or a slave becomes a problem. This is solved by software using BUF and M/S* bits of the ICW4 command.

BUF: 1 = 8259 used in buffered mode.
0 = 8259 used in non-buffered mode.
M/S*: 1 = 8259 is configured to function as master.
0 = 8259 is configured to function as slave.

The M/S* bit is a don't-care bit when BUF = 0.

Special fully nested mode: If the SFNM bit is 1, the 8259 will be configured for SFNM. This mode is meaningful only for Master 8259. To understand SFNM, it is necessary to first understand FNM.

Fully nested mode: The 8259 is configured to function in FNM if SFNM bit = 0 in the ICW4 command. If ICW4 command is not used in the configuring of 8259, the 8259 is automatically configured for FNM. In this mode IR_0 will have the highest and IR_7 will have the lowest priority to start with. This priority changes dynamically as interrupt requests arrive, if the 8259 is configured for rotating priority mode. Rotating priority is already described. If the 8259 is not configured for rotating priority, IR_0 will continuously have the highest and IR_7 the lowest priority.

The FNM of operation is normally used when there are no Slave 8259s. In FNM, IR_0 will have the highest and IR_7 the lowest priority if rotating priority is not used. Thus if IR_4 causes the activation of INT output, bit 4 of ISR will be set to 1. In such a case, IR_{4-7} cannot activate INT output till bit 4 of ISR is reset to 0 using EOI command with the help of OCW2 or using AEOI command with the help of ICW4 command.

Assume a Slave 8259 is connected to IR_4 input of the Master 8259 and the IR_6 input of Slave 8259 is responsible for the activation of INT output of the Slave. This is shown in Fig. 23.8.

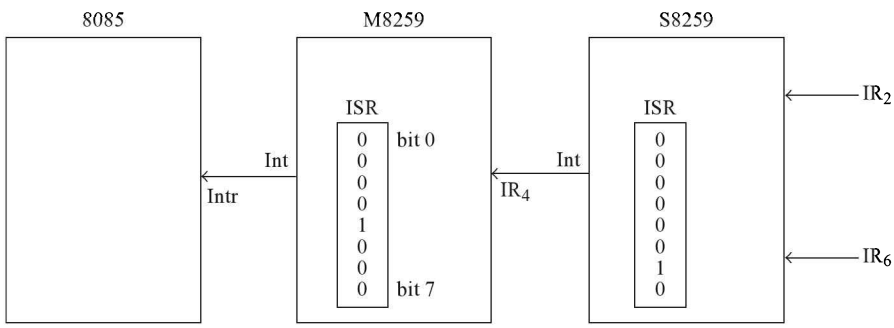


Fig. 23.8 Need for SFNM

As the INT output of Slave 8259 is connected to IR_4 input of Master 8259, bit 4 of ISR in the Master 8259 is set to 1. Now assume that the IR_2 input of Slave 8259 becomes active in the service routine for IR_6 of Slave 8259. As IR_2 has higher priority over IR_6 that is currently being serviced, the Slave 8259 again activates its INT output. But the Master 8259 does not activate its INT output because it is only the IR_4 input of Master 8259 that is active and bit 4 of ISR in the Master 8259 is not yet reset to 0. This problem is solved using the SFNM for Master 8259 and configuring Slave 8259s in FNM.

The SFNM is very similar to FNM with the following difference. Assume the processor is servicing an IR input of a Slave 8259. If a higher priority IR input of the Slave 8259 becomes active, the Master 8259 activates its INT output if the Master is configured for SFNM. This would not have happened if the master were in FNM. When the Master 8259 is configured for SFNM, the procedure to be adopted before exiting a service routine is as follows. It is explained with the situation described here as an example.

First of all, non-specific EOI command using OCW2 has to be issued to the Slave 8259. This would reset bit 2 of the ISR of Slave 8259. Then the ISR of Slave 8259 is read by issuing OCW3 command to the Slave 8259. If the contents of ISR of Slave 8259 is 00H, it means none of the IR inputs of Slave 8259 are currently being serviced. Only in such a case, non-specific EOI command using OCW2 has to be issued to the Master 8259. If the ISR contents of Slave 8259 is non zero, EOI command should not be issued to the Master 8259.

The Master 8259 is configured for SFNM if SFNM bit is set to 1 in the ICW4 command. If this bit is 0, the Master 8259 is configured for FNM only as indicated in the following.

SFNM: 1 = 8259 configured for SFNM.
0 = 8259 is not configured for SFNM. It is only FNM.

Default settings for ICW4: If the ICW4 command is not used in the system, the 8259 will automatically be configured as if the LS 5 bits of ICW4 command were 00000. In other words when ICW4 command is not used, the 8259 is automatically configured as follows.

- The processor used in the system is assumed to be 8085/8080.
- End of interrupt command needed before the end of the service routine.
- Non-buffered mode is used.
- Fully nested mode is used.

23.7 USE OF 8259 IN AN 8086-BASED SYSTEM

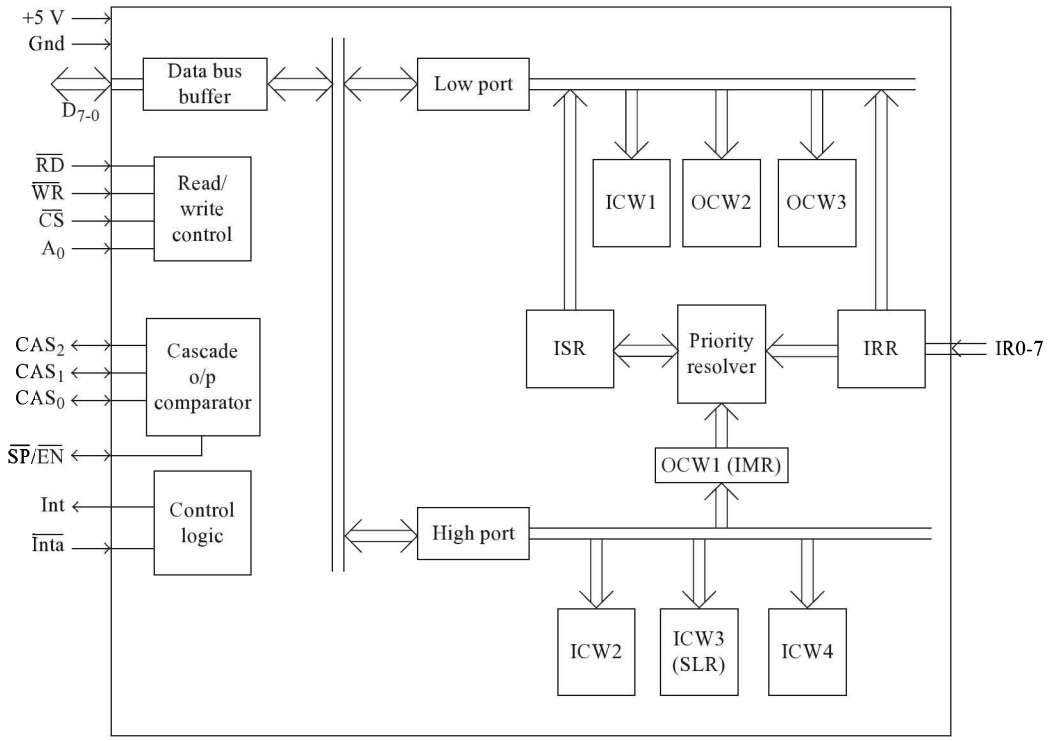
When 8259 is used in an 8086-based system, the 8259 automatically changes its working to suit the requirements of 8086 processor. The 8086 processor has two interrupt pins—INT and NMI. The INT output of 8259 is always connected to the INT input of 8086. It is never connected to NMI input of 8086. When the 8086 is interrupted on INT halfway through an instruction, the 8086 completes the instruction under execution and then sends out INTA* pulse two times in succession. When INTA* is pulsed the first time, the highest priority ISR bit is set to 1 and the corresponding IRR bit is reset to 0. The 8259 does not send out any data in response to the first INTA* pulse. In response to the second INTA* pulse, the 8259 sends out an 8-bit interrupt type number. It is to be noted that the 8259 does not send out a CALL instruction to the 8086. The 8086 performs branch to appropriate service routine based on the interrupt type number received by it.

When 8259 is used in an 8086-based system, some differences exist in the interpretation of ICW1 and ICW2 command words. The other commands have the same interpretation immaterial of whether the processor is 8085 or 8086.

ICW2 is used to provide the 8086 with the interrupt type number. In this command only the MS 5 bits are meaningful. The LS 3 bits are don't cares. Normally 000 is stored in these bit positions. The 8259 appends the IR input number that is responsible for the activation of INT output to the MS 5 bits of ICW2. If ICW2 command is issued with 10101 000 = A8H as the data, then it means that the interrupt type number is A8H if IR_0 input is responsible for the activation of INT output of 8259. If the IR_1 input is responsible for the activation of the INT output of 8259, the 8259 automatically sends out 10101 001 = A9H as the interrupt type number. Finally, if the IR_7 input is responsible for activation of INT output of 8259, the 8259 automatically sends out 10101 111 = AFH as the interrupt type number. In ICW1 command A_7, A_6, $A_{5/X}$, and 4/8* bits are don't-care bits when 8259 is used in an 8086-based system.

23.8 ARCHITECTURE OF 8259

Based on the description of the working of 8259, the simplified architecture of 8259 can be illustrated as in Fig. 23.9.



Bits 4, 3 of low port for selection of commands

Bit 4	Bit 3	Command
0	0	OCW2
0	1	OCW3
1	X	ICW1

Fig. 23.9 Internal architecture of 8259

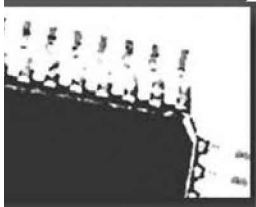
To conclude, the 8259 is a highly programmable chip that can be configured to exactly suit the requirements of interrupt control in an application.

1. Describe the need for an interrupt controller in a microcomputer system.
2. Provide a brief overview of the working of 8259 programmable interrupt controller.
3. Briefly describe the functions of the pins of 8259.
4. Describe the functions of the important registers available in 8259.

5. Provide an overview of the initialization command words used when a single 8259 is used in a system.
6. Provide an overview of the operation command words of 8259.
7. Is it possible to have an IV address for IR_0 as 3456H when the 8259 is configured for 4-byte CALL interval? If the answer is no, what is the possible IV address for IR_0 that is closest to 3456H?
8. Name the command word and indicate the contents of the command word that is required to configure the 8259 for the following operations.

   a. ICW4 is going to be issued.
   b. Several 8259s are to be used in the system.
   c. The address interval is required to be 4 bytes.
   d. The $IR_{0\text{-}7}$ requests are to be level triggered.
   e. The LS byte of IV address for IR_0 to be C0H.

9. Name the command word and indicate the contents of the command words that are required to configure three 8259s such that the Slave 8259s are connected on IR_2 and IR_5 inputs of Master 8259.
10. Name the command word of 8259 that specifies the processor used in the system. What should be the contents of this command word to configure the 8259 as buffered-mode slave, in FNM, with highest priority ISR bit getting automatically reset when the second INTA* pulse is received in an 8086-based system?
11. What is the change in the interpretation of bits in ICW1 and ICW2 commands of 8259 when the processor used in the system is changed from 8085 to 8086?
12. How does the 8259 identify the command written to the high port of 8259?
13. Explain the need for SFNM. How do you configure 8259 for SFNM?
14. Explain OCW2 command with relevant examples.
15. Explain polled mode of operation in 8259.
16. Explain SMM of 8259 and its utility.
17. What is the default setting of ICW4 command?
18. Explain the architecture of 8259 with a neat diagram.



24 Intel 8257—Programmable DMA Controller

- Concept of direct memory access (DMA)
- Need for DMA data transfer
- Description of 8257 DMA controller chip
  - *Condition when processor is the master and 8257 is in slave mode*
  - *Condition when processor is in the HOLD state and 8257 is in master mode*
- Programming the 8257
  - *Address registers*
  - *Count registers*
  - *Control register (mode set register) of 8257*
  - *Status register of 8257*
- Description of the pins of 8257
- Working of the 8257 DMA controller
- State diagram of 8085
- Questions

This chapter deals with the programmable DMA controller—Intel 8257. It gives a detailed description of the structure and working of 8257 with neat diagrams. The various aspects of 8257 are provided under several sections as mentioned in the outline for this chapter.

24.1 CONCEPT OF DIRECT MEMORY ACCESS (DMA)

In a microcomputer system there are basically three blocks: the microprocessor, memories such as RAM and EPROM, and I/O ports to which I/O devices are connected. Then the possible data transfers as indicated in Fig. 24.1 are as follows:

- Data transfer between microprocessor and memory (Ex. using LDA and STA instructions);
- Data transfer between microprocessor and I/O ports (Ex. using IN and OUT instructions);
- Data transfer between memory and I/O ports (DMA data transfer).

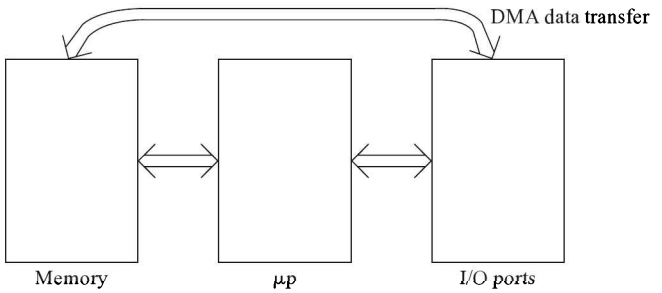


Fig. 24.1
Possible data transfers in a microcomputer system

For data transfer between microprocessor and memory or between microprocessor and I/O ports, instructions of the microprocessor are executed. As such, these data transfers are termed as programmed data transfer, which is generally used when a small amount of data transfer is involved.

Both microprocessor and memory are semiconductor chips that work at electronic speeds. Hence data transfer between microprocessor and memory is generally not a problem. In case memory is a bit slower than the microprocessor, the processor has to insert wait states. There is no problem other than this.

The I/O ports like 8255, 8212, and so on are also semiconductor chips that work at electronic speeds. However, they are used for interfacing I/O devices that are electromechanical in nature, and hence very slow compared to the speed of the microprocessor. As a result data transfer between microprocessor and I/O ports becomes more complex. In the chapter on 8255 the basic I/O, status check I/O, and the interrupt driven I/O schemes for data transfer between microprocessor and I/O ports are already discussed.

Data transfer between memory and I/O port directly (without going through the microprocessor) is called direct memory access (DMA). There are no instructions in the instruction set of the processor to perform DMA data transfer. The need for DMA data transfer and methods to perform it are discussed next.

■ 24.2 NEED FOR DMA DATA TRANSFER

If programmed data transfer is used for reading from memory location 3456H and writing to output port number 50H, it takes 13 clocks for reading from memory location 3456H using LDA 3456H instruction and ten clocks to write to output port number 50H. Thus it takes a total of 23 clocks. If the processor is working at 3-MHz internal frequency with a clock period of 0.33 μs, it takes 7.66 μs. Similarly, for reading from input port 40H and writing to memory location 2345H, it takes 7.66 μs using programmed data transfer.

If DMA data transfer is used, reading from memory location 3456H and writing to output port number 50H requires only four clocks, which amounts to only 1.33 μs. Similarly, for reading from input port 40H and writing to memory location 2345H, it takes 1.33 μs using DMA data transfer.

Some I/O devices like the hard disk and the floppy disk are capable of performing data transfers at quite a fast rate. If we have a 1.44-MB floppy diskette rotating at 360 rpm, and has 18 sectors per track, each sector storing 512 bytes, then the data transfer rate becomes 54K bytes per second, or about 19 μs per byte. Hard disks can easily transfer data at least ten times faster. Hence it turns out to

be 1.9 μs per byte. This is the situation when DMA data transfer becomes a must. In programmed data transfer, which needs about 7.66 μs per byte of data transfer, 4 bytes would have come out of the hard disk in the same time. Thus with fast I/O devices, DMA data transfer becomes a necessity.

Sometimes we may want to read from the A/D converter, say 1,000 times, and store the converted values in 1,000 memory locations. This may be needed to obtain statistics like the average, the largest, and the smallest, of the converted values. Using programmed data transfer, it takes $1{,}000 \times 7.66 = 7{,}660$ μs of processor time. Using DMA data transfer it takes only $1{,}000 \times 1.33 = 1{,}330$ μs. Thus DMA data transfer is desirable, if not essential, for data transfer between a slow I/O device (like A/D converter) and memory, if large amount of data transfer is desired.

Thus it is to be noted that when large amount of data has to be transferred between memory and I/O port, routing each byte via microprocessor becomes a time consuming operation. If the I/O port can directly access memory for data transfer, without processor intervention, it will be more efficient. Such a scheme is known as DMA data transfer.

However, as already mentioned, there are no instructions in the instruction set of a processor to perform DMA data transfer. If DMA data transfer has to take place without processor intervention, there must be a controller circuit on the I/O port that supervises DMA data transfer. Such a controller must have the following features.

- IOR, IOW, MR, and MW control signals generation capability.
- Memory address register to generate memory address for data transfer.
- Count register to indicate the number of bytes still to be transferred.

Generally I/O ports, like 8255 PPI chip, do not possess these features. Also, if more than one I/O port needs to perform DMA data transfer, then all these I/O ports need to have such controller circuitry. To solve this problem, Intel has developed programmable DMA controller chips like Intel 8257 and Intel 8237. Intel 8257 is described in this chapter.

■ 24.3 DESCRIPTION OF 8257 DMA CONTROLLER CHIP

Intel 8257 is a 40-pin programmable IC available as a DIP package. Its physical and functional pin diagrams are indicated in Figs. 24.2 and 24.3, respectively.

The 8257 works in two modes—slave mode and the master mode. Similarly, the processor also works in two modes—active mode and the HOLD mode. The processor is normally in the active mode. In the active mode, the processor is the master of the computer system, including the 8257. Only when DMA transfer is required to be performed, the processor goes to the HOLD state and gives up control of the system bus. In such a state the processor is logically disconnected from the rest of the computer system, and the 8257 becomes the master for the rest of the computer system.

The 8257 is in the slave mode when the processor is programming the 8257, or when the processor is reading the contents of the internal registers of 8257. At this point of time the processor is in active mode, and is the master of the computer system, including the 8257.

The 8257 is in the master mode when the 8257 is actually controlling DMA data transfer. At this point of time the processor is in the HOLD state. Hence 8257 is the master of the computer system, excluding the processor, which is logically disconnected in the computer system.

The 8257 can be used to control DMA data transfer of as many as four I/O ports. For each I/O port there is a corresponding DMA channel. Each DMA channel has its DMA request (DRQ) input, and a corresponding DMA acknowledge (DACK*) output. In addition, each DMA channel has a 16-bit address register (AR), and a 16-bit count register (CR).

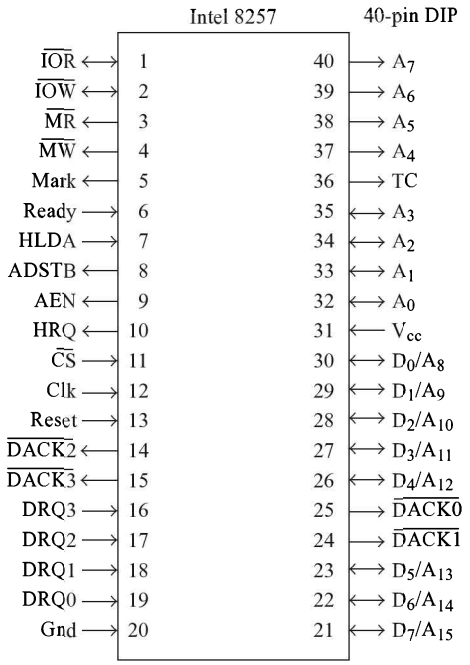


Fig. 24.2
Physical pin diagram of Intel 8257

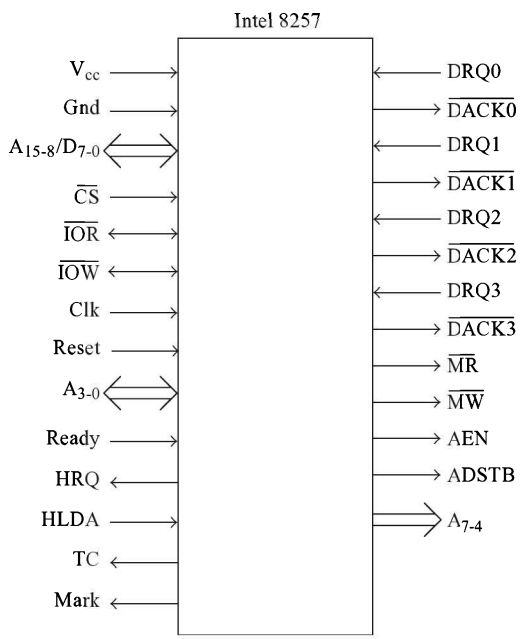


Fig. 24.3
Functional pin diagram of Intel 8257

The 8085 processor has two lines dedicated for DMA data operation. They are HOLD and HLDA (HOLD acknowledge). If an I/O port is in need of DMA service, it activates a DRQ input of the 8257, which in turn sends out HRQ (HOLD request) on HOLD input of 8085. The 8085 then completes the current machine cycle (note: not the current instruction cycle) and then goes to HOLD state. In the HOLD state the address pins, data pins, RD*, WR*, and IO/M* pins are tristated. Thus the 8085

is effectively disconnected from the rest of the system, which is made known by activating HLDA output of the 8085.

The 8057, which was so far a slave to the 8085, and was receiving commands from the 8085, now becomes the master of the computer system. Of course, it does not become the master of 8085, which is logically disconnected from the computer system. The 8257 resolves the priorities of the requesting I/O ports (if there is more than one DMA request active), and sends DACK* signal to the highest priority I/O port needing DMA service.

24.3.1 Condition when Processor is the Master and 8257 is in Slave Mode

D_{7-0}/A_{15-8} are used as bi-directional data lines for communication between the processor and an internal register of 8257.

A_{3-0} are input lines of 8257, to select an internal register of 8257 for communication with the processor.

IOR* and IOW* are input lines of 8257 so that the processor can read from or write to the internal registers of 8257.

MR*, MW*, and A_{7-4}, which are the output pins of 8257, are tristated by 8257.

24.3.2 Condition when Processor is in the HOLD State and 8257 is in Master Mode

D_{7-0}/A_{15-8} are used as uni-directional address output lines for sending out MS byte of address from an AR in 8257.

A_{3-0} are output lines of 8257 and are used to send out the LS 4 bits of address from an AR in 8257.

A_{7-4} are output lines of 8257 and are used to send out the MS 4 bits of LS byte of address from an AR in 8257.

IOR*, IOW*, MR*, and MW* are output pins of 8257. If the required operation is DMA read machine cycle, MR* and IOW* signals will be activated by the 8257. The IOR* and MW* signals will be in the inactive state. If the required operation is DMA write machine cycle, IOR* and MW* signals will be activated by the 8257. The MR* and IOW* signals will be in the inactive state.

■ 24.4 PROGRAMMING THE 8257

The programmer's view of 8257 is provided in Fig. 24.4. From the microprocessor point of view, it is an I/O port chip that is used exclusively for DMA control application. It is not used for interfacing I/O devices for the purpose of data transfer with the processor. This chip can be used to control DMA data transfer of as many as four I/O ports. For each I/O port there is a corresponding DMA channel. The chip provides all the features needed for a DMA data transfer.

24.4.1 Address Registers

For each DMA channel there is an address register (AR) and a count register (CR). These registers are 16-bits wide. Thus, there are four ARs, AR_{3-0}, each of 16 bits. Similarly there are four CRs, in addition to control and status registers. The control and status registers are separate 8-bit registers, but having

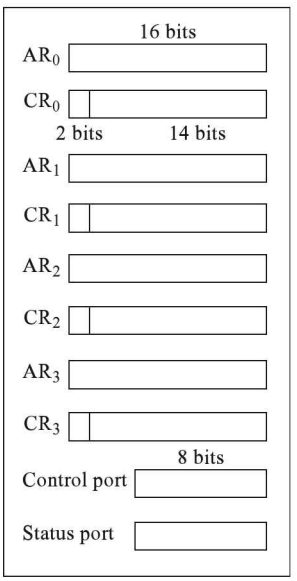


Fig. 24.4
Programmer's view of Intel 8257

the same address. The control register can only be written, and the status register can only be read by the processor.

To select one of these registers, four address pins are needed. $A_{3\text{-}0}$ address pins of 8257 are used for this purpose. The processor writes to an AR, CR, or control register by sending an appropriate address on $A_{3\text{-}0}$ pins when chip select (CS*) and I/O write (IOW*) signals are active. The processor reads from an AR, CR, or status register by sending an appropriate address on $A_{3\text{-}0}$ pins when CS* and I/O read (IOR*) signals are active.

An AR contains the address of a memory location that is used in DMA data transfer. The AR used in a DMA transfer is automatically incremented by 1 after every DMA access. As there are only eight pins for communication with the processor, the LS and MS bytes of an AR are written in two steps, starting with the LS byte. Thus, when the processor writes to an AR for the first time, the LS byte of the address register is written, and when it is written the second time, the MS byte is written.

First/last flip-flop: The 8257 contains a first/last (F/L) flip-flop which toggles after each access of an AR or CR. This flip-flop may be more appropriately called the MS/LS* (M/L*) flip-flop. The state of this flip-flop decides whether MS or LS bytes of an AR/CR is accessed.

If M/L* = 0, LS byte is accessed;
If M/L* = 1, MS byte is accessed.

This flip-flop is reset when the RESET input of 8257 is activated. It is also reset when the Control port is written by the processor. The LS and MS bytes should be accessed in succession. It is not proper, for example, to access AR_0 and AR_1 registers in the sequence indicated as follows.

Write to AR_0 ; LS byte of AR_0 is written
Write to AR_1 ; MS byte of AR_1 is written, when our intention was to write to LS byte of AR_1
Write to AR_0 ; LS byte of AR_0 is written again, when our intention was to write to MS byte of AR_0
Write to AR_1 ; MS byte of AR_1 is written again

Interrupts should be disabled prior to programming the registers of 8257. Otherwise, an interrupt could occur just after accessing the LS byte of a register and can cause problems.

24.4.2 Count Registers

There are four CRs, $CR_{3\text{-}0}$, each of 16 bits. When the processor accesses a CR that is 16-bits wide, the LS and MS bytes of the register are alternately accessed, starting with the LS byte. Again, the M/L* flip-flop helps in this requirement. The CR contains information about the number of bytes to be transferred using DMA. It is decremented by 1 for every byte of DMA data transfer. When the CR becomes 0, it performs the last DMA data transfer which results in the activation of the terminal count (TC) output by the 8257. There is only one TC output that is used by all the four channels. The processor reads the status port of 8257 to find out which channel was responsible for the activation of TC output by 8257.

Although a CR is 16-bits wide, only the LS14 bits of the register are used to specify the number of bytes to be transferred using DMA. So a maximum of 2^{14} = 16K = 16,384 bytes can be programmed for data transfer. If N is the number of bytes to be transferred using DMA data transfer, we have to load $(N-1)$ in the LS 14 bits of the CR. If the CR is loaded with 0 value in the LS 14 bits, only one byte will be transferred.

The MS 2 bits of a CR specify the type of DMA transfer. It can be DMA read (read from memory and write to I/O port), DMA write (read from I/O port and write to memory), or DMA verify. The meaning of the MS 2 bits of a CR are indicated as follows, if the 8257 is connected as I/O-mapped I/O device. In such a case, the IOR* and IOW* signals generated by the processor are connected to IOR* and IOW* input pins of 8257.

Bit 15	*Bit 14*	*Operations*
0	0	DMA verify
0	1	DMA write
1	0	DMA read
1	1	Illegal

In the above case, when bit 15 is 1, the 8257 generates MR* and IOW* signals. When bit 14 is 1, it generates MW* and IOR* signals.

If desired, 8257 can be connected as a memory-mapped I/O device in a system. In such a case, the MR* and MW* signals generated by the processor are connected to IOR* and IOW* pins of 8257. The meaning of the MS 2 bits of a CR are changed as indicated in the following, if the 8257 is connected as memory-mapped I/O device.

Bit 15	*Bit 14*	*Operations*
0	0	DMA verify
0	1	DMA read
1	0	DMA write
1	1	Illegal

In the above case, when bit 15 is 1, the 8257 generates IOR* and MW* signals. When bit 14 is 1, it generates IOW* and MR* signals.

In DMA verify mode, no control signal is activated by 8257. Hence actual data transfer does not take place. It is normally used after some data have been transferred to an I/O port by memory using DMA read operation. It verifies that the information transferred to the I/O port is a true copy of the

data in memory. For this purpose, the 8257 sends out DACK* signal for each DMA verify cycle. The peripheral device may use these signals to verify the data it has acquired. It calculates the CRC (cyclic redundancy check) character for the stored data on I/O device and compares with the stored value of the CRC on the device to verify the data stored.

24.4.3 Control Register (Mode Set Register) of 8257

The processor, when active, writes to the Control register of 8257 to configure its working. To find out the status of 8257, the status register is read by the processor. The control register is 8-bits wide, and can only be written by the processor, and not read by it. It is selected when CS* = 0, $A_{3\text{-}0}$ = 1000, and IOW* = 0.

The control register of 8257 is normally called the mode set register by Intel, as it sets the mode of operation of 8257. The meaning of the contents of the control register is indicated in Fig. 24.5.

AULD	TCS	EXWR	RTPR	ECH3	ECH2	ECH1	ECH0

AULD: 1 = Enable auto load
0 = Disable auto load

TCS: 1 = Stop DMA transfer if TC reached
0 = No automatic stop of DMA on occurrence of TC

EXWR: 1 = Enable extended write
0 = Disable extended write

RTPR: 1 = Enable rotating priority
0 = Disable rotating priority

ECH*n*: 1 = Enable DMA Channel *n* (*n* = 0–3)
0 = Disable DMA Channel *n*

Fig. 24.5
Interpretation of the contents of the control register

Enabling/disabling of DMA channels: The HRQ output of the 8257 is activated when a DRQ input is activated, and the corresponding DMA channel has been enabled. Thus, if a DMA channel is disabled, the HRQ output is not going to be activated even if the corresponding DRQ input is active. The contents of ECH3-0 bits in the control register decide the enabling or disabling of the DMA channels. After reset of 8257, all the four DMA channels are disabled.

Rotating priority: The 8257 uses fixed priority for the servicing of DMA requests after reset input of 8257 is activated. Channel 0 has the highest priority and Channel 3 has the lowest priority. Thus the priority for the channels in descending order would be 0, 1, 2, 3. In such a case, if DMA Channel 0 is enabled and the DRQ0 request is active, then the DMA channel 0 data transfer starts. If the I/O device connected to DRQ0 does not withdraw its request till all bytes are transferred, no other channel can be serviced till Channel 0 is fully serviced. Such a data transfer is called burst mode data transfer.

If all the channels are to be serviced with equal priority, the rotating priority has to be selected by setting to 1 the RTPR bit in the control register. In such a case, the DMA channel, which has just been serviced, by transferring a byte using DMA, is allotted the lowest priority. Highest priority is allocated to the channel with the next higher number. The channel numbers 0, 1, 2, 3 can be considered to form a circle. For example, if Channel 2 is serviced just then, it gets the lowest priority and Channel 3 the highest priority. Thus the priority for the channels in descending order would become 3, 0, 1, 2.

Similarly, if Channel 1 is serviced just then, it gets the lowest priority and Channel 2 the highest priority. Thus the priority for the channels in descending order would become 2, 3, 0, 1. If all the four DMAs are enabled, and all the four DRQ inputs are active, then each DMA channel gets its chance to perform a DMA data transfer in four DMA cycles.

Extended write: A DMA machine cycle in which a byte is transferred directly between memory and I/O port needs four clock cycles.

- In the first clock cycle it will be in state S1. In this state, address is sent out from the relevant AR by 8257. MS byte is sent out on A_{15-8}/D_{7-0} and LS byte on A_{7-0}.
- In the second clock cycle it will be state S2. In this state, MR* (for DMA read machine cycle) or IOR* (for DMA write machine cycle) is activated.
- In the third clock cycle it will be state S3. In this state, IOW* (for DMA read machine cycle) or MW* (for DMA write machine cycle) is activated.
- In the fourth clock cycle it will be state S4. In this state, a byte of data is transferred between selected memory location and I/O port.

In the S3 state, the ready input is checked by the 8257. This ready input is similar in function to the ready input in 8085. Only if this input is a 1, the 8257 goes to state S4 during the next clock. If this input is a 0, the 8257 enters a wait state. It comes out of wait state and goes to S4 only after ready input becomes 1.

In a DMA verify machine cycle there will not be any wait states and hence it needs only four clock cycles. This is because there will not be any data transfer in a DMA verify machine cycle.

In a DMA read or write machine cycle, if the extended write bit is set, the IOW* (for DMA read machine cycle) or the MW* (for DMA write machine cycle) is generated in the second clock cycle itself. This provides for extended write times. This helps in avoiding wasteful wait states to a great extent.

Stop on terminal count (TCS bit): When the LS 14 bits of a DMA CR is decremented to zero value, the TC output of 8257 is activated. If the TCS bit in the Control register is set to 1, then the corresponding DMA channel is automatically disabled (corresponding ECH bit of control port reset to 0) on the activation of the TC output. This prevents further DMA operation on that channel. The ECH bit for the channel must be set to 1 to begin another DMA operation.

If the TCS bit in the control port is reset to 0, the activation of TC output has no effect on the DMA channel. It is then the responsibility of the I/O port to withdraw DMA request to terminate the DMA operation.

Auto load: Auto load feature is useful for repeat block operations and block chaining operations. This feature makes use of Channel 2 only, with help always from Channel 3, described as follows. Thus, in auto load mode Channel 3 is not available to the user for DMA data transfer. Hence the ECH3 bit is reset to 0 to disable Channel 3.

Repeat block operation: Assume that we want to display a page of information on the CRT. The CRT uses raster scan to display the information, which has to be refreshed about 50 times per second to get a flicker-free display of the page. The information in a page of display RAM can be sent to CRT controller using DMA read machine cycles.

Let us say, we want to transfer 1000H bytes starting from display RAM location 6000H to CRT controller. Then AR_2 can be loaded with 6000H and the MS 2 bits of CR_2 with 10 to signify DMA read machine cycle. The LS 14 bits of CR_2 are loaded with 1000H that is 01 0000 0000 0000. Thus CR_2 is loaded with 9000H. When the CRT controller activates DRQ2 input of 8257, DMA read

machine cycles start. TC output is activated when CR_2 contents are decremented to 8000H (when all the LS 14 bits of CR_2 become 0s). At this point of time, the AR_2 contents would have become 7000H.

Now it is necessary to reload AR_2 with 6000H and CR_2 with 9000H to repeat the sequence, and this has to be done 50 times per second to get flicker-free display. This operation is automated using the auto load feature.

If the AULD bit in the control port is set to 1, then the values stored in Channel-2 registers are automatically loaded into Channel-3 registers. Note that the auto load feature makes use of only Channel 2 with help from Channel-3 registers. Thus when AR_2 and CR_2 are loaded with 6000H and 9000H, AR_3 and CR_3 are also automatically loaded with 6000H and 9000H. When the DRQ2 request input is activated by the CRT controller, the DMA read machine cycles start. After 1000H bytes are transferred, AR_2 and CR_2 would have become 7000H and 8000H, and TC output would be activated. At this point of time, the Channel-2 registers are automatically reloaded from Channel-3 registers. This happens during an 'update' cycle, when the UD (update) flag in the status register is set to 1 by the 8257. Then the values in Channel-3 registers are copied to Channel-2 registers. The UD bit is reset to 0 by the 8257 only after DMA transfer starts again on Channel 2.

Even if the TCS bit is set to 1, it will not disable Channel 2 when TC output is activated, if AULD bit is set to 1. Hence the block operations are automatically repeated if DRQ2 input remains active.

Block chaining operation: The maximum number of bytes that can be transferred using DMA data transfer is 2^{14} = 16K = 16,384. What if we want to transfer the contents of memory from 6000H to 7400H, which is 17K bytes, to an I/O port? In such a situation block chaining operation can be used as described in the following.

First of all, AULD bit is set to 1, ECH2 bit to 1, and ECH3 bit is reset to 0 in the control port. Then load AR_2 with 6000H and CR_2 with 9000H, as described for block repeat operation. These values are automatically loaded into Channel-3 registers during a UD cycle. The UD bit in the status register is monitored till it becomes 0. Then the AR_3 is loaded with 7000H and CR_3 with 10 00 0100 0000 0000 = 8400H. Now the DMA transfer for Channel 2 takes place and the TC output is activated when AR_2 has become 7000H and CR_2 8000H. This would have transferred 1000H bytes out of our requirement of 1400H. At this point, the channel registers are reloaded from Channel-3 registers. Thus AR_2 becomes 7000H and CR_2 becomes 8400H. Now the next block of DMA transfer for Channel 2 results in the transfer of 400H bytes. The TC output is activated when the AR_2 is 7400H and CR_2 8000H. With the block chaining scheme, any amount of information can be transferred using DMA data transfer.

24.4.4 Status Register of 8257

The status register is 8-bits wide, and can only be read but not written by the processor. It is selected when CS* = 0, $A_{3\text{-}0}$ = 1000, and IOR* = 0.

The status register of 8257 provides status information about the present state of 8257. Only the LS 5 bits are meaningful in the status register as indicated in Fig. 24.6.

0	0	0	UD	TC3	TC2	TC1	TC0

UD: 1 = Update in progress
0 = Update not in progress

TCn: 1 = TC reached for Channel n (n = 0–3)
0 = TC not yet reached for Channel n

Fig. 24.6
Interpretation of the contents of the status register

The UD flag bit is not affected by a read operation of the status register. The UD flag is cleared to 0 under the following conditions.

- Intel 8257 is reset;
- Auto load feature disabled by resetting to 0 the AULD bit in the control register;
- Upon completion of UD cycle.

The TC_{3-0} bits are cleared to 0 under the following conditions.

- Intel 8257 is reset;
- Status register is read by the processor.

24.5 DESCRIPTION OF THE PINS OF 8257

The description of the pins of 8257 is given in Fig. 24.3. The reader is advised to read this portion once again after the programming of 8257 is explained.

Vcc and Gnd: Power supply and ground pins. 8257 uses +5V power supply.

D_{7-0}/A_{15-8}: Used as eight bi-directional data pins for communication with the processor, when the processor is active and the 8257 is in the slave mode. When the processor is in the HOLD state and the 8257 is the master, they are used to send out the MS 8 bits of memory address from an AR of 8257.

A_{3-0}: When the processor is active, they are used as address input pins of 8257 to select one of the registers inside the 8257. There are ARs, CRs, control and status registers as shown in the following table. When the processor is in the HOLD state, these pins are used to output the LS 4 bits of memory address by the 8257.

A_{3-0}	*Selected register*
0000	AR_0
0001	CR_0
0010	AR_1
0011	CR_1
0100	AR_2
0101	CR_2
0110	AR_3
0111	CR_3
1000	Control/status
1001 to 1111	Unused

A_{7-4}: When the processor is in the HOLD state, these pins are used to output the MS 4 bits of the LS byte of memory address by the 8257. These pins are tristated, when the processor is active.

RESET: This is an active high input pin which is normally connected to the RESET OUT pin of 8085. As such, when the 8085 is reset, it sends out a logic 1 pulse on the RESET OUT pin thus resetting the 8257. After the reset of 8257, the control register contents becomes 00H. This means:

- All the four DMA channels are disabled.
- Uses fixed priority. Channel 0 will have highest and Channel 3 the lowest priority.
- Extended write option is disabled.

- When the TC is reached for a DMA channel, the channel is not automatically disabled. Hence the responsibility of terminating a DMA request rests with the I/O device.
- Auto load feature is disabled.

DRQ3-0: These are active high DMA request input pins, one for each of the four DMA channels. They are activated by special-purpose I/O port chips like Intel 8272—floppy disk controller, and Intel 8275—CRT controller. In fixed priority mode (when RTPR bit is 0 in the control register), DRQ0 has the highest, and DRQ3 the lowest priority.

DACK3-0*: These are active low DMA acknowledgement output pins, one for each of the four DMA channels. Logical OR of chip select circuit output and DACK* output of 8257 is given as chip select input to special purpose I/O ports like Intel 8272 and Intel 8275. A DACK* output becomes 0 and then 1 for each byte of DMA data transfer.

IOR*: It is an active low input pin that is activated by the processor to read an AR, CR, or the status register, when the 8257 is in the slave mode. The IOR*, generated using IO/M*, RD*, and WR* signals of 8085, is connected to this pin when 8257 is connected as I/O-mapped I/O device. If the 8257 is connected as memory-mapped I/O device, the MR*, which is generated using IO/M*, RD*, and WR* signals of 8085, is connected to this pin.

When the processor is in the HOLD state, this becomes an output pin, and is activated for a DMA write machine cycle.

IOW*: It is an active low input pin that is activated by the processor to write to an AR, CR, or the control register, when the 8257 is in the slave mode. The IOW*, generated using IO/M*, RD*, and WR* signals of 8085, is connected to this pin when 8257 is connected as I/O-mapped I/O device. If the 8257 is connected as memory-mapped I/O device, the MW*, which is generated using IO/M*, RD*, and WR* signals of 8085, is connected to this pin.

When the processor is in the HOLD state, this becomes an output pin, and is activated for a DMA read machine cycle.

MR*: It is an active low output pin that is in tristate when the 8257 is in the slave mode. When the processor is in the HOLD state, the 8257 drives this pin. It is activated for a DMA read machine cycle and is inactive for a DMA write machine cycle.

MW*: It is an active low output pin that is in tristate when the 8257 is in the slave mode. When the processor is in the HOLD state, the 8257 drives this pin. It is activated for a DMA write machine cycle and is inactive for a DMA read machine cycle.

CS*: Active low input pin used for selecting the chip.

Clk: It is the clock input pin. The maximum allowed clock frequency on this input is about 3 MHz. The clock input is connected to the ClkOut pin of 8085 in an 8085-based system.

Ready: This is an active high input pin. It is similar in function to the ready input of 8085. Devices with slow access times can use this input to insert wait states during DMA read or write machine cycles. The wait states are never inserted in the case of DMA verify machine cycles.

HRQ: HRQ stands for HOLD request. This is an active high output pin. This is connected to the HOLD input of 8085. Whenever a DRQ input is active, and the corresponding DMA channel is enabled, the HRQ output is activated by the 8257. It then essentially requests the processor to grant control of the system bus.

HLDA: HLDA stands for HOLD acknowledge. This is an active high input pin, which is connected to the HLDA output of 8085. When the HLDA input becomes active, it means that the processor has gone to HOLD state and has relinquished the system bus. Now the 8257 becomes the master in the microcomputer system.

TC: TC stands for terminal count. This is an active high output pin. This output is activated when all the LS 14 bits of the CR become 0 for the DMA channel being serviced. There is a single TC output pin although there are four DMA channels. If the I/O ports connected to the DMA channels have TC input pin, then the TC output of 8257 is connected to the TC input pin of the I/O port. For example, the TC output of 8257 is connected to the TC input of Intel 8272—floppy disk controller chip. Thus when the TC output is activated, the I/O port for which DACK* is active, receives the TC input. If the TCS bit in the control port of 8257 is reset to 0, it will be the responsibility of the I/O port to withdraw the DRQ. If the TCS bit in the control port of 8257 is set to 1, the DMA channel being serviced is automatically disabled. However, if Channel 2 is being serviced, and AULD bit in the control port is set to 1, the Channel 2 remains enabled irrespective of the TCS bit value.

MARK: This is an active high output pin. This output is activated when the LS 7 bits of the CR become 0 for the DMA channel being serviced. In other words, whenever the LS byte of the CR becomes 80H or 00H. This happens whenever the number of bytes still to be transferred is an integral multiple of 128 (80H). There is a single MARK output pin although there are four DMA channels. This output is not normally used in a microcomputer system, because, the chips that use DMA data transfer, like 8272 and 8275, do not have MARK input.

AEN: AEN stands for address enable. This is an active high output pin. Intel 8257 outputs a 0 on AEN whenever 8085 is the master of the computer system. When the 8085 goes to the HOLD state, the 8257 outputs a 1 on AEN, indicating that 8257 is the master now.

ADSTB: ADSTB stands for address strobe. This is an active high output pin which is similar in function to the ALE output of 8085. Intel 8257 outputs a 0 on this pin as long as it is in slave mode. When the 8257 becomes the master, it outputs 1 on ADSTB only during the first of the four clock cycles of a DMA machine cycle.

Fig. 24.7 illustrates the interfacing of 8257 in an 8085-based system. It uses two Intel 8212 non-programmable I/O port chips and a quad 2-to-1 multiplexer. Both 8212s are used in mode 0. At any instant of time only one of the two 8212s is selected. This is because the AEN output of the 8257 is connected to DS1* input of 8212-1 and DS2 input of 8212-2.

Thus 8212-1 chip is selected when 8257 outputs a 0 on AEN in the slave mode. The 8212-2 chip is selected when the 8257 outputs a 1 on AEN in the master mode. It is easy to note from Fig. 24.7 that the 8085 generates the address and control signals MR*, MW*, IOR*, and IOW* when the AEN output of 8257 is 0. These control signals are generated by the 2-to-1 multiplexer using RD*, WR*, and IO/M* outputs of 8085.

MR* output is activated only when RD* is 0 and IO/M* is 0
MR* output is deactivated when RD* is 1 or IO/M* is 1

MW* output is activated only when WR* is 0 and IO/M* is 0
MW* output is deactivated when WR* is 1 or IO/M* is 1

IOR* output is activated only when RD* is 0 and IO/M* is 1
IOR* output is deactivated when RD* is 1 or IO/M* is 0

IOW* output is activated only when WR* is 0 and IO/M* is 1
IOW* output is deactivated when WR* is 1 or IO/M* is 0

Intel 8257 generates the address and control signals MR*, MW*, IOR*, and IOW* when the AEN output of 8257 is 1. These control signals are directly generated by the 8257 on its pins. The ADSTB output is pulsed to latch the MS byte of address sent out by the 8257.

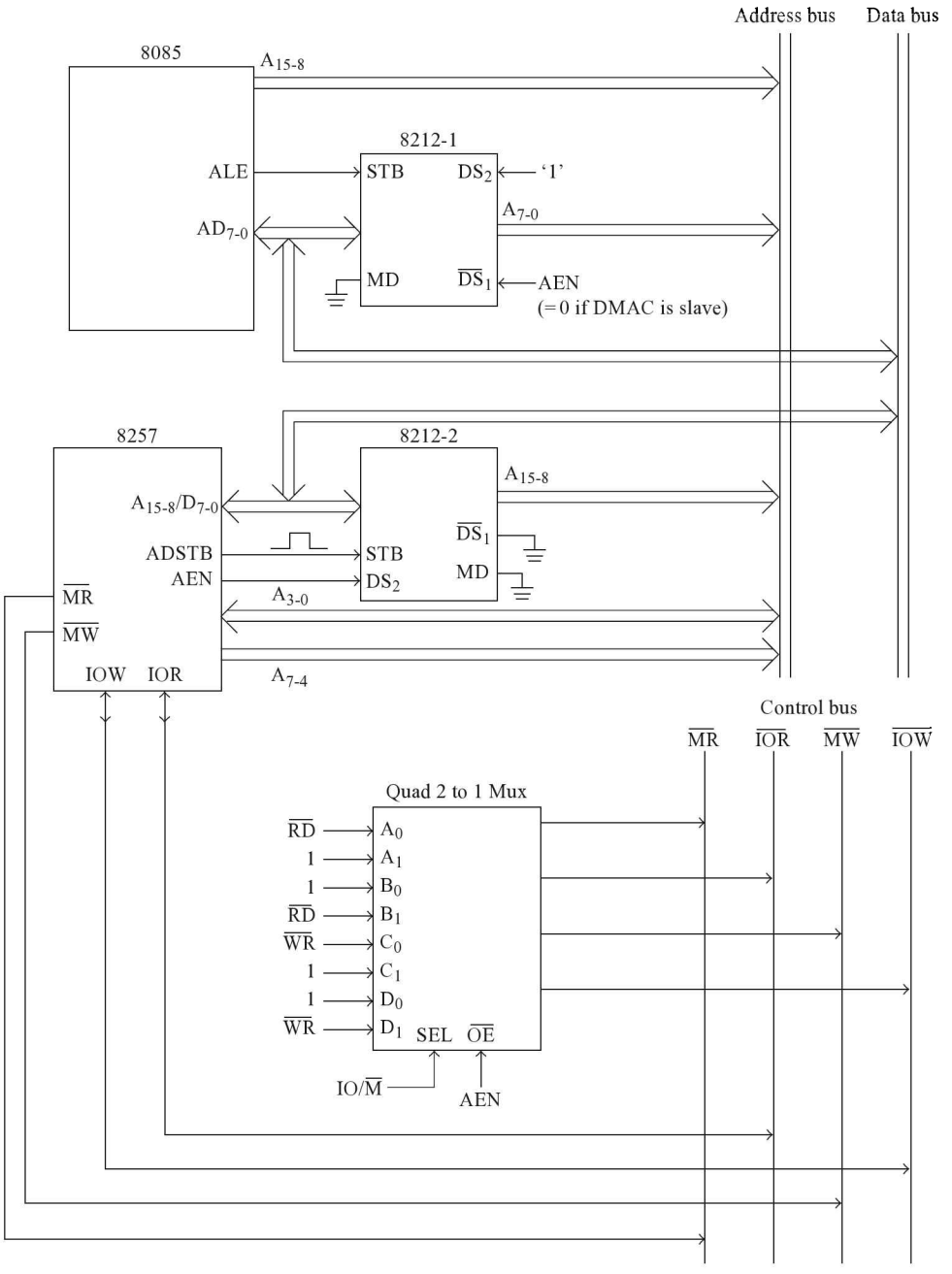


Fig. 24.7 Interfacing of 8257 in an 8085-based system

■ 24.6 WORKING OF THE 8257 DMA CONTROLLER

First, the 8257 is programmed by the processor. At this point, the processor is the master and the 8257 works in the slave mode. The processor programs each channel by writing to the AR the starting address of memory for data transfer, and writing to the CR the number of bytes to be transferred using the DMA. The number of bytes information is specified using LS 14 bits of the CR. The MS 2 bits indicate the type of DMA data transfer. Then the processor writes to the control port (referred to as mode set register by Intel). This specifies the DMA channels that are enabled, whether it is fixed or rotating priority, and the like.

Whenever an I/O port is in need of DMA service, it activates the DRQ input of a channel. The 8257 in turn activates the HRQ. The HRQ output of 8257 is connected to the HOLD input of 8085. Thus a DMA request activates the HOLD input of 8085. The 8085 then completes the *current machine cycle* (not the current instruction cycle) and then goes to the HOLD state. In the HOLD state the address pins, data pins, RD*, WR*, and IO/M* pins are tristated. Thus the 8085 is effectively disconnected from the rest of the system. It announces that it is in the HOLD state by activating the HLDA output.

The 8057, which was so far a slave to the 8085, and was receiving commands from the 8085, now becomes the master of the computer system. Of course, it does not become the master of 8085, which is logically disconnected from the computer system. The 8257 resolves the priorities of the requesting I/O ports (if there is more than one DMA request active), and sends DACK* signal to the highest priority I/O port needing DMA service. This results in chip selection of the requesting I/O device. The 8257 sends out the 16-bit memory address present in AR using the pins $A_{15\text{-}8}/D_{7\text{-}0}$, $A_{7\text{-}4}$, and $A_{3\text{-}0}$.

If the required operation is DMA read machine cycle, then the MR* and IOW* signals will be activated by the 8257 and the IOR* and MW* signals will be in the inactive state. If the required operation is DMA write machine cycle, then the IOR* and MW* signals will be activated by the 8257 and the MR* and IOW* signals will be in the inactive state. The outputting of memory address and the activation of control signals results in the transfer of a byte of data between the memory and I/O port directly. This DMA machine cycle takes four clock cycles. At the end of each DMA machine cycle, the CR is decremented by 1, the AR is incremented by 1, and DACK is deactivated.

There are three types of DMA data transfers:

- Single-byte transfer;
- Short-burst mode;
- Long-burst mode.

Single-byte transfer: Some I/O ports like Intel 8272 generate a DMA request for each byte of DMA data transfer. The following steps are performed for each byte of data transfer.

- Intel 8272 activates the DRQ input of 8257.
- Intel 8257 activates the HOLD input of 8085, by activating HRQ output.
- Intel 8085 enters the HOLD state, suspends program in execution, and activates the HLDA output.
- Intel 8257 activates the DACK* output.
- Intel 8257 generates the required control signals to perform data transfer of a byte. The AR is incremented and CR is decremented.
- Intel 8272 deactivates the DRQ request.
- Intel 8257 deactivates the HOLD input of 8085, by deactivating the HRQ output.
- Intel 8085 comes out of the HOLD state and deactivates the HLDA output.
- Intel 8085 resumes suspended program for some time.
- Intel 8272 reactivates the DRQ and the sequence repeated till TC is reached.

This kind of a data transfer is called *Single byte transfer*, as the DRQ is activated for each byte of data transfer. It is also termed *Cycle stealing* data transfer in the literature. This is because the 8257 steals a DMA machine cycle in the middle of an instruction cycle, when a byte of DMA is needed. This scheme is used with slow peripherals.

Short-burst mode: Some I/O ports like Intel 8275 generate a DRQ and keep it active till several bytes are transferred. The DRQ input of the 8257 is kept active depending on how the 8275 is programmed, for upto eight DACK* activations. This results in 8 bytes of DMA data transfer. Then the DRQ output remains inactive for quite some time. During this period, the HRQ output of 8257 becomes inactive resulting in 8085 coming out of HOLD state and getting into active state. Then the sequence mentioned earlier is repeated till all the bytes are transferred and the TC output is activated. This kind of a data transfer is called *Short-burst mode transfer*, as the DRQ input remains active till a small block of data is transferred. This scheme is used with moderately fast peripherals.

Long-burst mode: In this mode, the DRQ request is withdrawn by the I/O port only after all the bytes are transferred and the TC output is activated. But the DACK* output is pulsed for each byte of data transfer. This scheme is used with fast peripherals like hard disk controller.

■ 24.7 STATE DIAGRAM OF 8085

By now we have fully discussed about the pins, the instruction set, and the various machine cycles of 8085. Now we study the state diagram of 8085, as illustrated in Fig. 24.8.

Intel 8085 starts in T_{reset} state at power on. At this point the following flip-flops will have the status as shown in the following.

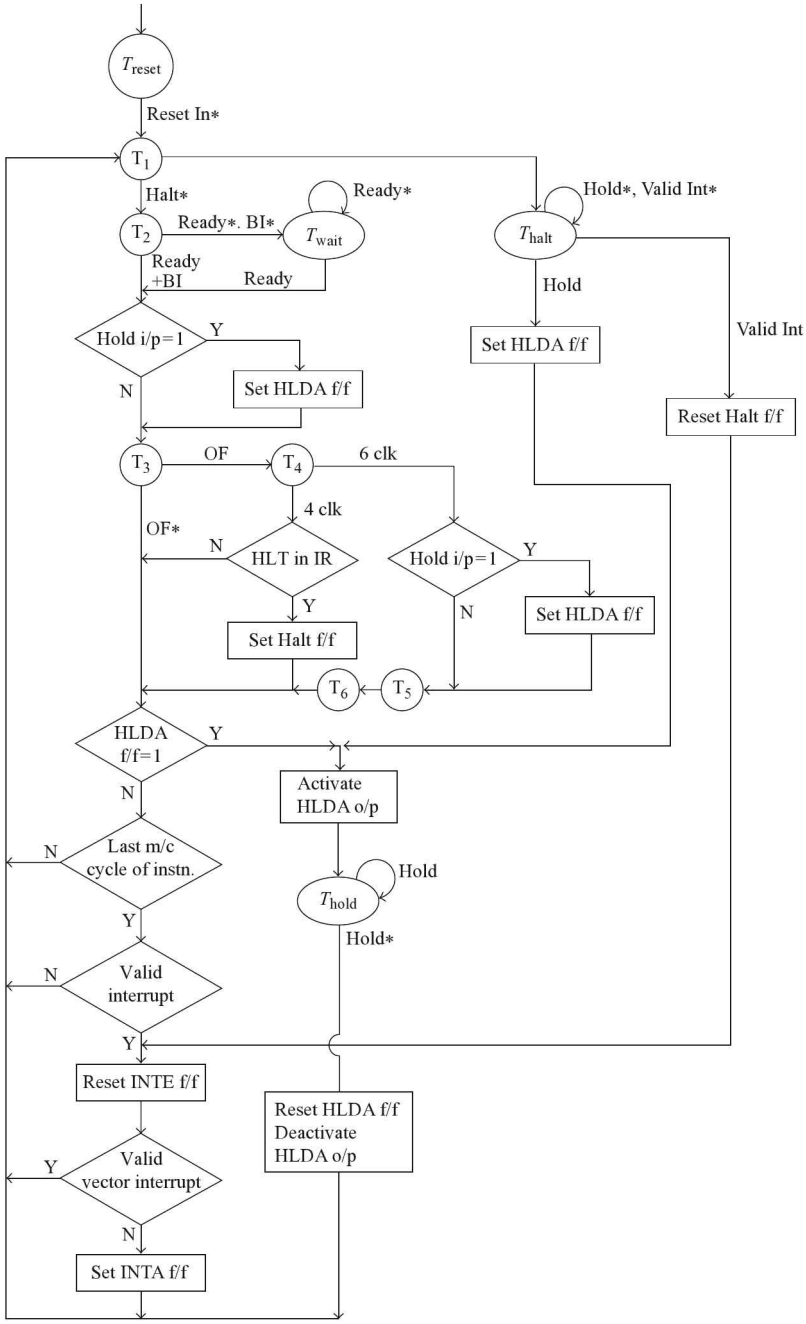
- Halt flip-flop reset to 0.
- The HLDA flip-flop is reset to 0. It results in the HLDA output of 8085 becoming 0.
- The INTE flip-flop is reset to 0. It disables the RST 7.5, RST 6.5, RST 5.5, and INTR.
- The INTA flip-flop is reset to 0. It results in the INTA* output of 8085 becoming 1.

The halt flip-flop is set to 1 when the 8085 decodes the instruction in IR as HLT instruction.

Intel 8085 checks the value of the HOLD input pin at the end of T2 in a machine cycle consisting of three or four clock cycles. It is checked at the end of T4 also, if the machine cycle consists of six clock cycles, like the first machine cycle of a CALL instruction. If it senses a 1 on the HOLD input, the HLDA flip-flop is set to 1. In such a case, at the end of the *current machine cycle*, not necessarily at the end of the instruction cycle, the 8085 activates the HLDA output and enters T_{hold} state.

Intel 8085 checks all the interrupt request pins at the end of the last clock cycle in an *instruction cycle*. If it senses any valid interrupt request the INTE flip-flop is reset to 0. This results in disabling of all interrupts except TRAP so that the ISS can be executed without further interrupts. If the 8085 has been interrupted because of INTR and none of the higher priority interrupt requests are active, then the INTA flip-flop is set to 1. This results in the activation of INTA* output of 8085.

When reset-in becomes inactive, the 8085 enters the state T1. This is the first clock cycle of a machine cycle. At the end of T1, if the HALT flip-flop status is 0, it enters state T2. At the end of T2, if the ready input of 8085 is 0 and the machine cycle is not a BI machine cycle, then the 8085 enters T_{wait} state. (Intel 8085 executes a BI machine cycle only for DAD instruction or when it is required to respond to a vector interrupt.) As long as the ready input remains 0, the 8085 remains in T_{wait} state. If the ready input becomes 1, then the 8085 comes out of T_{wait} state.



Notes

- No wait states in BI m/c cycle.
- It is a valid interrupt case when
  - Trap is activated;
  - INTR is activated when INTE f/f = 1;
  - RST 7.5/6.5/5.5 activated when INTE f/f = 1 and the interrupt pin is not masked.
- 8085 comes out of T_{halt} if there is a valid interrupt.
- 8085 responds to Hold i/p even in T_{halt}.
- In such a case, it returns to T_{halt} after the DMA data transfer.

Fig. 24.8 State diagram of 8085

At the end of T_2, if the ready input of 8085 is 1 or the machine cycle is a BI machine cycle, then the 8085 checks the HOLD input. If the HOLD input is not active, then the 8085 enters T_3 state. If the HOLD input is active, then the 8085 sets the HLDA flip-flop, and enters the T_3 state.

If it is the first machine cycle of an instruction (OF machine cycle), the 8085 enters T_4 state from T_3. At the end of T_4, if it is the last clock cycle for the machine cycle, 8085 checks if the code for HLT instruction is in the IR. If yes, the 8085 sets the HALT flip-flop, and then checks the status of the HLDA flip-flop. If the IR is not having the code for HLT, the 8085 directly checks the status of the HLDA flip-flop.

At the end of T_4, if it is a six-clock machine cycle, the 8085 again checks the HOLD input. If the HOLD input is not active, the 8085 enters T_5 state. If the HOLD input is active, the 8085 sets the HLDA flip-flop, and then enters T_5 state. From state T_5 it enters state T_6. At the end of T_6 the HLDA flip-flop status is checked.

If it is not the first machine cycle of an instruction (like MR, MW, IOR, IOW, INA, BI machine cycles), the HLDA flip-flop status is checked at the end of the T_3 state. Thus the HLDA flip-flop status is checked at the end of:

- T_3 for MR, MW, IOR, IOW, INA, and BI machine cycles;
- T_4 for OF machine cycle with four-clock cycles;
- T_6 for OF machine cycle with six-clock cycles.

If the HLDA flip-flop is set, the 8085 activates the HLDA output pin and enters T_{hold} state. It remains in the T_{hold} state as long as the HOLD input is active. When the HOLD input is deactivated, the 8085 comes out of the HOLD state and resets the HLDA flip-flop. Then the 8085 goes ahead with the first clock cycle for the next machine cycle in the instruction by entering state T_1.

If at the end of T_3 (T_4 or T_6 as the case may be) the HLDA flip-flop is in reset condition, then the 8085 finds out if it is the last machine cycle for the current instruction. If it is not the last machine cycle, the 8085 begins the next machine cycle by entering T_1 state again. If it is the last machine cycle, it means the current instruction execution is complete. Then it checks for any valid interrupts. If there are no valid interrupt requests, the 8085 goes ahead with the next instruction cycle by entering T_1 state again.

If there is any valid interrupt at the end of an instruction cycle, the 8085 resets the INTE flip-flop. This ensures that the execution of ISS can proceed without any further interrupts. Then 8085 checks to find out if INTR is the only valid interrupt. If yes, it sets the INTA flip-flop that activates INTA* output pin. Then the 8085 goes ahead with receiving a 3-byte CALL instruction, or a 1-byte RST*n* instruction from an I/O device, by entering T_1 state of an INA machine cycle. If an interrupt other than INTR is a valid interrupt, the 8085 directly goes ahead with BI machine cycle, by entering T_1 state again.

At the end of T1, if HALT flip-flop status is 1, the 8085 enters state T_{halt}. If the HOLD input remains inactive and there is no valid interrupt request, the 8085 remains in T_{halt} state. It will wait in this state for the activation of the HOLD input or of a valid interrupt request.

If the HOLD input becomes active during T_{halt}, the 8085 sets the HLDA flip-flop, and activates the HLDA output pin and enters T_{hold} state. As discussed earlier, the 8085 remains in the T_{hold} state as long as the HOLD input is active. When the HOLD input is deactivated, it comes out of the HOLD state and resets the HLDA flip-flop. Then the 8085 re-enters T_{halt} state from T_1 state.

If valid interrupt becomes active during T_{halt}, the 8085 resets the HALT, and INTE flip-flops. This ensures that the 8085 has come out of halt state, and the execution of ISS can proceed without any further interrupts. Then, as discussed earlier, the 8085 checks to find out if INTR is the only valid interrupt. If yes, it sets the INTA flip-flop that activates the INTA* output pin. Then the 8085 goes ahead with receiving a 3-byte CALL instruction, or a 1-byte RST*n* instruction from an I/O device, by entering T_1 state of an INA machine cycle. If an interrupt other than INTR is a valid interrupt, the 8085 directly goes ahead with BI machine cycle, by entering T_1 state again.

1. What is meant by DMA? What is the need for DMA data transfer?
2. Describe the need for a DMA controller in a microcomputer system.
3. Provide a brief overview of the working of 8257 DMA controller.
4. Briefly describe the functions of the pins of 8257.
5. Explain the function of $A_{3\text{-}0}$, MR*, MW*, IOR*, IOW*, and $D_{7\text{-}0}/A_{15\text{-}8}$ pins of 8257 when it is working as a slave to the processor.
6. Explain the function of $A_{3\text{-}0}$, MR*, MW*, IOR*, IOW*, and $D_{7\text{-}0}/A_{15\text{-}8}$ pins of 8257 when it is working as a master controlling DMA data transfer.
7. Describe the meaning of the MS 2 bits of a count register, when 8257 is connected as a memory-mapped I/O device.
8. Describe the meaning of the MS 2 bits of a CR, when 8257 is connected as an I/O-mapped I/O device.
9. Explain auto load mechanism feature of 8257.
10. With a neat diagram explain the interfacing of 8257 in an 8085-based system.
11. Describe the function of the important registers available in 8257.
12. With an example, describe the meaning of every bit in the control port of 8257.
13. With an example, describe the meaning of every bit in the status port of 8257.
14. With a neat diagram describe the state transition diagram of 8085 processor.
15. Write the control word needed to perform the following:
    a. Enable Channels 0, 3 and disable Channels 1, 2.
    b. Enable rotating priority and extended write, disable auto load and TC stop.
16. Assume 8257 control port contains 53H. How does 8257 react to this?
17. Assume 8257 status port contains 53H. What does it mean?
18. Explain single byte transfer, short burst mode, and long burst mode data transfers.

25

Intel 8253—Programmable Interval Timer

- Need for programmable interval timer
- Description of 8253 timer
- Programming the 8253
  - *Read on the fly*
  - *Internal architecture of a counter*
- Mode 0—interrupt on terminal count
- Mode 1—re-triggerable mono-stable multi
- Mode 2—rate generator
- Mode 3—square wave generator
- Mode 4—software-triggered strobe
- Mode 5—hardware-triggered strobe
- Use of 8253 in ALS-SDA-85 kit
- Questions

This chapter deals with Intel 8253, which is a programmable interval timer. The various sections mentioned in the outline of this chapter provide a detailed description of Intel 8253.

25.1 NEED FOR PROGRAMMABLE INTERVAL TIMER

There are many situations where accurate time delays are required to be generated in a microcomputer system. For example, if we are implementing a real time clock, the time has to be updated once every second.

Accurate time delays can be generated using a few instructions in a loop as explained in Sect. 17.13. This is the software method, in which the 8085 does not do any useful work except generating time delay.

Time delays could be generated by hardware also. For example, a timer chip like 555 could be used to generate time delays. In this case, the time delay generated will be dependent on the resistor and capacitor component values. These R and C components will typically have 10% tolerance. Thus, the time delay generated will not be very precise, but the processor is now free to do other processing.

In order to get accurate time delays by hardware method, programmable timer chips like Intel 8253 are used. In this method, there is very little software overhead and the processor is available for other processing.

■ 25.2 DESCRIPTION OF 8253 TIMER

Intel 8253 is a 24-pin programmable IC available as a DIP package. It has three independent counters each of 16-bits width. In addition, there is a control port to decide the mode of working of the three counters. Its physical and functional pin diagrams are indicated in Figs. 25.1 and 25.2, respectively.

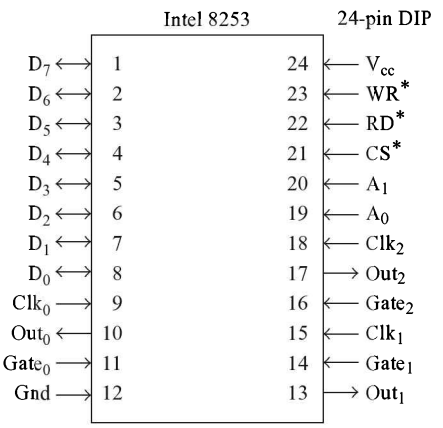


Fig. 25.1
Pin diagram of Intel 8253

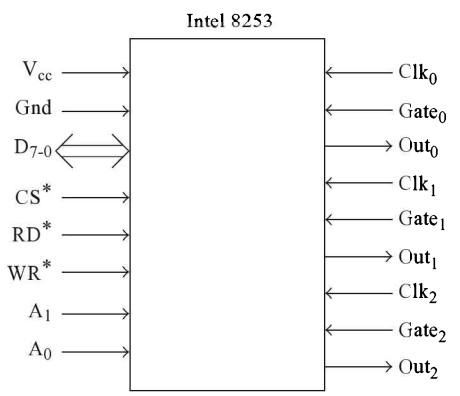


Fig. 25.2
Functional pin diagram
of Intel 8253

V_{cc} and Gnd: Power supply and ground pins. 8253 uses +5V power supply.
$D_{7\text{-}0}$: Eight bi-directional data pins for communication with the processor.
RD*: Active low input pin that is activated by the processor to read counter information from the 8253.
WR*: Active low input pin that is activated by the processor to write counter and control information to the 8253.
CS*: Active low input pin used for selecting the chip.

A_1, A_0: Address input pins. They are used along with RD*, WR*, and CS* to select one of the counters or the control port. The control port can be written only from the processor. The processor cannot read it. The counters can be read or written. Table 25.1 provides the details.

Table 25.1 Selection of a counter port or control port

A_1	A_0	RD*	WR*	CS*	Operation
0	0	0	1	0	*Read Counter0
0	1	0	1	0	*Read Counter1
1	0	0	1	0	*Read Counter2
1	1	0	1	0	**No operation
0	0	1	0	0	***Write to Counter0
0	1	1	0	0	***Write to Counter1
1	0	1	0	0	***Write to Counter2
1	1	1	0	0	Write to control port
X	X	X	X	1	**No operation
X	X	1	1	0	**No operation

Notes
*Read LS byte, or MS byte, or LS and then MS byte, or LS and then MS byte on the fly.
**$D_{7\text{-}0}$ will be tristated.
***Write to LS byte, or MS byte, or LS and then MS byte. There is no write on the fly.

CLK_0: Provides clock input for Counter0.
CLK_1: Provides clock input for Counter1.
CLK_2: Provides clock input for Counter2. Maximum frequency allowed for any of these clocks is 2 MHz.
$Gate_0$: It is an input pin that controls the function of Counter0.
$Gate_1$: It is an input pin that controls the function of Counter1.
$Gate_2$: It is an input pin that controls the function of Counter2. The gate inputs have different effects in different modes on the functioning of the corresponding counters.
Out_0: It is an output pin on which Counter0 sends its output.
Out_1: It is an output pin on which Counter1 sends its output.
Out_2: It is an output pin on which Counter2 sends its output. The output generated by the timer depends on the mode of operation.

■ 25.3 PROGRAMMING THE 8253

From the point of view of a microprocessor, the 8253 is a specialized I/O port chip. It is never used for interfacing I/O devices. It is only used for timing applications in a microcomputer. The 8253 has A_1 and A_0 as the address input pins. Thus only four addresses are possible for the 8253 ports as seen from a microprocessor, as was shown in Table 25.1.

The counters are however 16 bits in width. This is because, if they were 8-bits wide, the time delay generated would have been very small. The LS byte and the MS byte of a counter is selected using the same port address. When it is desired to load both the LS and MS bytes of a counter: the first time the port is written, it is stored in the LS byte, and the next time the port is written, it is stored in the MS byte. This way, the same port address is used to access both the LS and the MS bytes of a counter.

The processor writes to the control port to configure the working of the three timers. Actually, the processor has to write three times to the control port to configure the working of the three counters.

The contents of the control port convey the following information to the 8253.

- Selects a counter for configuration.
- Configures selected counter for a particular mode of operation.
- Decides whether the selected counter should count down in decimal or hexadecimal.
- Decides whether LS byte or MS byte, or LS and then the MS byte are to be loaded.

The counters can be configured to work in any of the following six modes of operation.

- Mode 0—interrupt on terminal count;
- Mode 1—re-triggerable mono-stable multi;
- Mode 2—rate generator;
- Mode 3—square wave generator;
- Mode 4—software-triggered strobe;
- Mode 5—hardware-triggered strobe.

The interpretation of the contents of the control port is as shown in Fig. 25.3.

SC_1	SC_0	RW_1	RW_0	M2	M1	M0	BCD

SC_1, SC_0:
00 = Counter0 selected
01 = Counter1 selected
10 = Counter2 selected
11 = Illegal

M2, M1, M0:
000 = Mode 0—interrupt on terminal count
001 = Mode 1—re-triggerable mono-stable multi
010 = Mode 2—rate generator
011 = Mode 3—square wave generator
100 = Mode 4—software-triggered strobe
101 = Mode 5—hardware-triggered strobe
110 = Illegal
111 = Illegal

BCD:
1 = Perform count down in decimal
0 = Perform count down in hexadecimal

RW_1, RW_0:
00 = Read on the fly for read operation. It is illegal for write operation
01 = Read/write LS byte of counter
10 = Read/write MS byte of counter
11 = Read/write LS byte and then MS byte of counter

Fig. 25.3 Interpretation of the contents of the control port

For the discussion in this chapter, the chip select circuit is assumed to be such that the port addresses are as follows.

Address	*Port*
80H	Counter0
81H	Counter1
82H	Counter2
83H	Control port

25.3.1 Read on the Fly

There are times when it is needed to read the counter value when the count down is in progress and take a decision based on the current counter contents. If it is desired to read the 16-bit contents of Counter2 and store in DE pair for later processing, then it is enough to execute the following instructions.

```
IN 82H;      Read LS byte of Counter2
MOV E, A;    Move it to E register
IN 82H;      Read MS byte of Counter2
MOV D, A;    Move it to D register
```

However, when these instructions are being executed, the counter goes ahead with the count-down operation too. This could lead to wrong reading of the counter value. For example, let us say the count value was 3400H when reading of the counter is attempted. Then, the E register gets the value 00H. But by the time the IN 82H instruction is executed the second time, the counter could have counted down to 33FFH. In such a case, the D register gets the value 33H. In other words, the counter value is wrongly interpreted to be 3300H instead of the correct 3400H.

To avoid the above mentioned problem, the count-down operation of the counter must be inhibited temporarily which could be done by inhibiting the clock input. In some modes, the count-down operation can be inhibited by making the corresponding gate input 0.

However, if it is desired to read the contents of Counter2 without affecting its normal count-down operation, the 'read on the fly' command is used. In such a case, the control port is written with the value 10 00 XXXX. It selects Counter2 for read on the fly operation. The Counter2 value, say 3400H is latched in a temporary register in 8253. Figure 25.4 provides the internal architecture of 8253, showing the presence of the temporary register. The count-down operation for Counter2 continues as usual. Thus DE now gets the correct 3400H. It is to be noted that the programmer always has to read the LS and MS bytes in the 'read on the fly' operation.

```
MVI A, 10XXXXXXB;
OUT 83H;     Set up Counter2 for read on the fly operation
IN 82H;      Read LS byte of Counter2
MOV E, A;    Move it to E register. E value is 00H
IN 82H;      Read MS byte of Counter2
MOV D, A;    Move it to D register. D value is 34H
```

25.3.2 Internal Architecture of a Counter

The internal architecture of a counter is extracted from Fig. 25.4 and is shown in Fig. 25.5.

It is to be noted that to each counter, there is a corresponding background counter, denoted as BCounter. The counter loads the BCounter at the appropriate moment depending on the mode of operation of the counter. The counter value remains unchanged even when clock pulses occur. It is

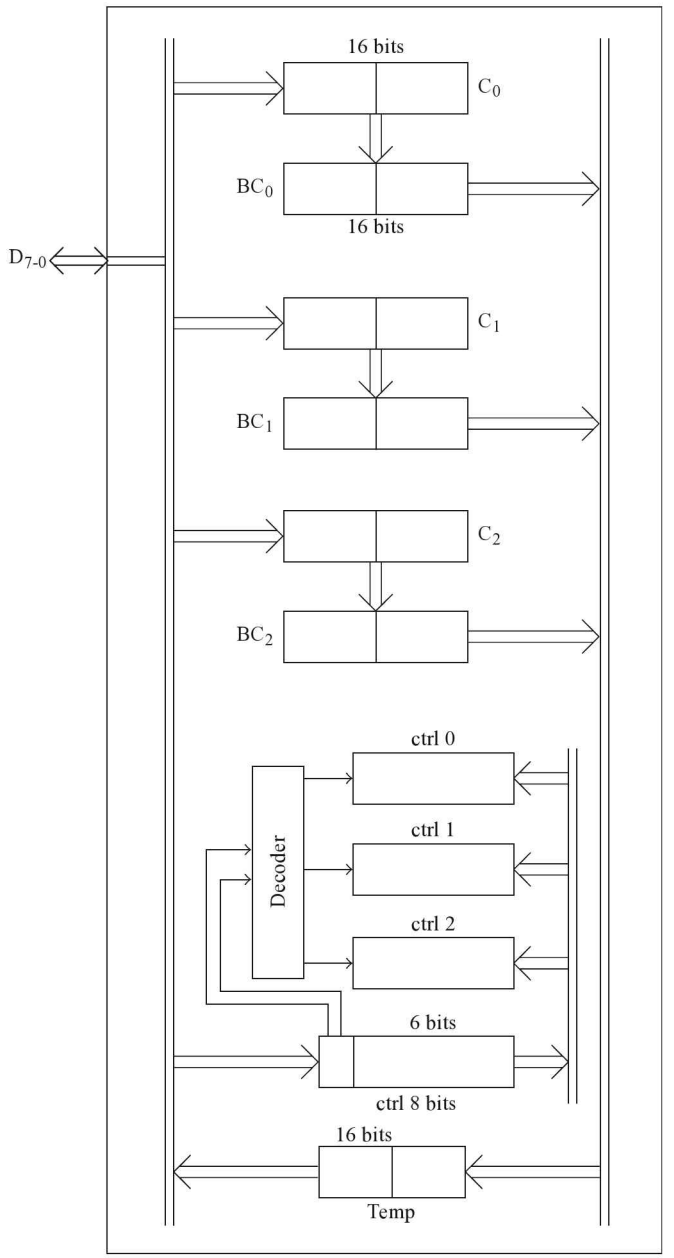


Fig. 25.4
Internal architecture of 8253

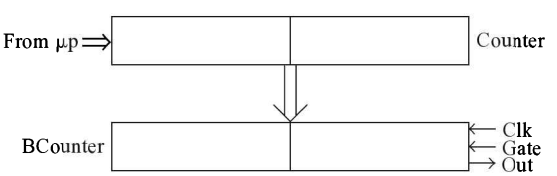


Fig. 25.5
Internal architecture of a counter

only the BCounter that is decremented for every clock pulse. BCounter is affected by the gate input depending on the mode of operation of the counter.

■ 25.4 MODE 0—INTERRUPT ON TERMINAL COUNT

Before a counter is set up for mode-0 (or any other mode) operation, the corresponding output pin will be in tristate. As soon a counter is set up for mode 0, the output will become logic 0. In mode-0 operation, the counter value is copied into the corresponding BCounter when the processor loads the counter a little later. The count down of BCounter takes place for every occurrence of clock pulse, as long as gate input is at logic 1. The count down of BCounter is complete when the BCounter value reaches the TC (Terminal Court) value of 0000H. Then the output pin goes to logic 1. This can be used to interrupt the processor. Hence this mode is known as 'interrupt on terminal count'.

If the gate input becomes logic 0 during the count-down operation, the count-down operation is suspended temporarily. When the gate input becomes logic 1 again, the count down continues from the point of suspension.

If the counter is loaded with a new value, say 5,234, when the count down of BCounter is in progress, then the output will go to logic 1 only after 5,234 clock pulses from the time the counter is loaded with the new value 5,234.

Let us say, it is desired to have Counter2 operate in mode 0 with a count value of 3,412 decimal. Then it is necessary to execute the following instructions.

MVI A, **10***11*0001**B**
OUT 83H; Set up Counter2 for Mode 0. Load LS and then MS bytes. Decimal count down

MVI A, 12H
OUT 82H; Load LS byte of Counter2 with 12.

MVI A, 34H
OUT 82H; Load MS byte of Counter2 with 34. Thus load Counter2 with 3412 decimal.

Figure 25.6 provides the output waveform when the gate input is at logic 1 throughout the count-down operation. If the Clock2 frequency is 1 MHz, the Out_2 pin becomes logic 1 after 3,412 μs.

Figure 25.7 provides the output waveform when the gate input becomes logic 0, when count down has reached a value of 2,400. When the gate input becomes logic 1 again a little later, the count down continues from 2,400. Thus, if the $gate_2$ input is in logic-0 state for 1,000 μs, the Out_2 pin goes to logic 1 after 4,412 μs.

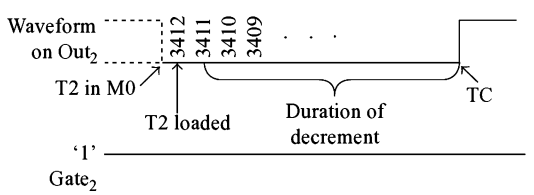


Fig. 25.6
Counter output waveform in mode 0 when gate input is at logic 1

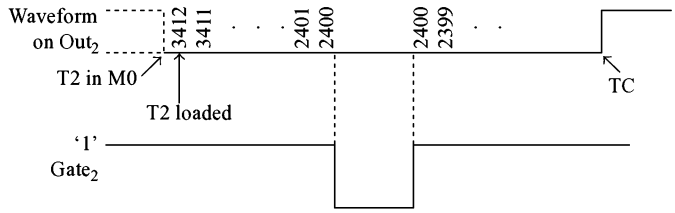


Fig. 25.7
Effect of gate input on counter output waveform in mode 0

■ 25.5 MODE 1—RE-TRIGGERABLE MONO-STABLE MULTI

The stable state of the multi-vibrator in this mode is logic 1. When a trigger pulse occurs on the gate input, the multi-vibrator output goes to logic 0 state, which is the quasi-stable state. It remains in this state for a duration dependent on the loaded counter value and the clock frequency. Then it reverts to the stable state of logic 1.

Before a counter is set up for mode 1 (or any other mode) operation, the corresponding output pin will be in tristate. As soon as a counter is set up for mode 1, the output will become logic 1, the stable state. The processor loads the counter a little later. However, the counter value is still not copied into the corresponding BCounter. The value in the counter is copied to the BCounter only when the trigger pulse occurs, in the form of gate input making a 0 to 1 transition. It results in the output changing to the quasi-stable state of logic 0. From then on, the count down of BCounter takes place for every occurrence of clock pulse, immaterial of the logic state of the gate input. In other words, 1 to 0 transition of the gate input has no effect on the count-down operation. The count down of BCounter is complete when the BCounter value reaches the TC value of 0000H. Then the output reverts to the stable state of logic 1.

If the gate input makes a 0 to 1 transition during the count-down operation, the count-down operation is restarted from the beginning. This is because the 0 to 1 transition on the gate input causes reloading of the BCounter from the counter. This re-triggering increases the duration of the quasi-stable state. Hence this mode is known as 're-triggerable mono-stable multi'.

Assume the counter is loaded with a new value, say 5,234, when the count down of BCounter is in progress. This will have no effect on the output as long as no 0 to 1 transition occurs on the gate input during the count-down operation. If there is a 0 to 1 transition on the gate input during the count down after the new value 5,234 is loaded, the count down restarts from the value 5,234. This re-triggering increases the duration of the quasi-stable state.

Let us say, it is desired to have Counter2 operate in mode 1 with a count value of 3,412 decimal. Then it is necessary to execute the following instructions.

```
MVI A, 10110011B
OUT 83H; Set up Counter2 for Mode 1. Load LS and then MS bytes. Decimal count down

MVI A, 12H
OUT 82H; Load LS byte of Counter2 with 12.

MVI A, 34H
OUT 82H; Load MS byte of Counter2 with 34. Thus load Counter2 with 3412 decimal.
```

Figure 25.8 provides the output waveform when there is no 0 to 1 transition on the gate input during the count-down operation. If the Clock2 frequency is 1 MHz, the Out_2 pin becomes logic 1 after remaining in the quasi-stable state for 3412 μs.

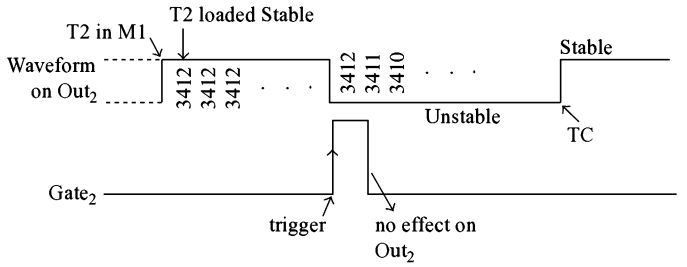


Fig. 25.8
Counter output waveform in mode 1 (without re-triggering)

Figure 25.9 provides the output waveform when the gate input provides another trigger pulse when the count down has reached a value of 2,412. The count down then restarts from 3,412. Thus, the quasi-stable state becomes longer, and the Out_2 pin goes to logic 1 after 4,412 μs.

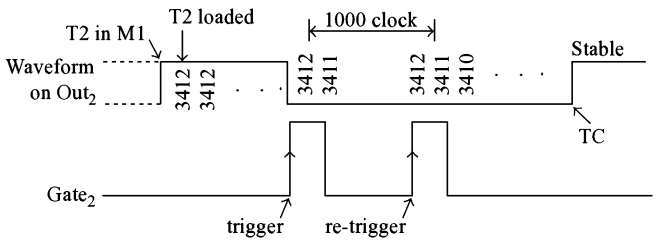


Fig. 25.9
Effect of re-triggering on counter output waveform in mode 1

■ 25.6 MODE 2—RATE GENERATOR

Before a counter is set up for mode 2 (or any other mode) operation, the corresponding output pin will be in tristate. As soon as a counter is set up for mode 2, the output becomes logic 1.

The BCounter is loaded from the corresponding counter when any of the following three conditions is satisfied.

- Gate = 1, counter is not presently running, but is loaded;
- Gate makes a 0 to 1 transition;
- Gate = 1, counter is presently running, and TC is reached.

In mode-2 operation, the output will remain high during the count-down process. The output becomes logic 0 for one input clock period only for the last count down. The BCounter is then reloaded from counter value, and the count-down operation continues with the output at logic 1. If the counter is loaded with a value of 1234H and the clock input has a period of 1 μs, then the output waveform in mode 2 will be an infinite series of pulses with a pulse width of 1 μs and the interval between pulses will be equal to 1,234 μs. Thus the output is the same as divide by 1234H of the input clock frequency. Hence mode 2 is called rate generator or frequency divider. In mode-2 operation it is possible to have the operation synchronized by software or by hardware.

In software synchronization, the gate input is held at logic 1 throughout. The moment the counter is loaded by the processor, the BCounter is loaded from the counter and the count-down operation starts. During the last count, the output becomes logic 0 for one input clock period. Then the BCounter is reloaded from counter value, and the count-down operation continues with the output at logic 1.

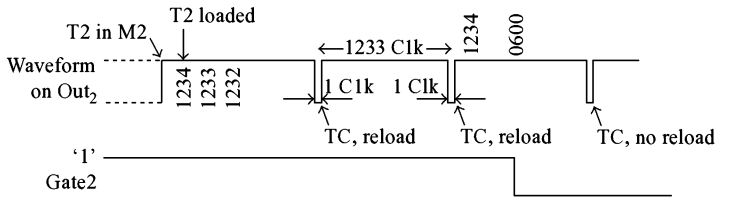


Fig. 25.10
Software synchronization
in mode 2

Later, even if the gate input is made logic 0, the count down continues and generates a pulse during the last count and the operation stops. This is shown in Fig. 25.10 where a count value of 1234H and clock input period of 1 μs are assumed.

In hardware synchronization, the gate input is held at logic 0 initially. Then, the BCounter is not loaded from the counter even when the processor loads the counter. Only when the gate input makes a 0 to 1 transition, the BCounter is loaded from counter and the count-down operation begins. During the last count, the output becomes logic 0 for one input clock period. Then the BCounter is reloaded from counter value, and the count-down operation continues with the output at logic 1. Later, even if the gate input is made logic 0, the count down continues and generates a pulse during the last count and the operation stops. This is shown in Fig. 25.11 where a count value of 1234H and clock input period of 1 μs are assumed.

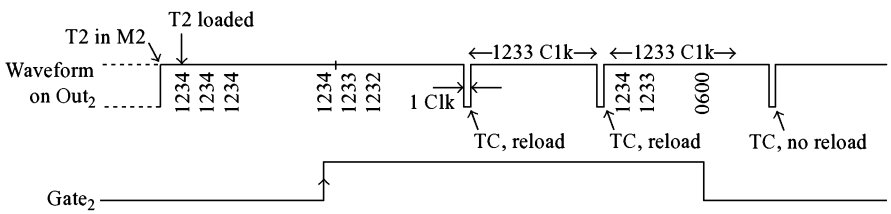


Fig. 25.11 Hardware synchronization in mode 2

Assume that the counter is presently operating in mode 2 with a count value of 1234H. Let us say the counter is loaded with a new value, 5,234 when the count down is in progress. This will not affect the count-down process. Thus after 1,233 clock pulses, the output goes to logic 0 for one clock period. The BCounter is then loaded from the counter with the new value of 5,234 and the count down starts again.

Let us say, it is desired to have Counter2 operate in mode 2 with a count value of 3,412 decimal. Then it is necessary to execute the following instructions.

MVI A, **10***11*0101**1**B
OUT 83H; Set up Counter2 for Mode 2. Load LS and then MS bytes. Decimal count down

MVI A, 12H
OUT 82H; Load LS byte of Counter2 with 12.

MVI A, 34H
OUT 82H; Load MS byte of Counter2 with 34. Thus load Counter2 with 3412 decimal.

■ 25.7 MODE 3—SQUARE WAVE GENERATOR

The mode-3 operation is similar to mode 2, except that square waves are generated instead of pulses. Before a counter is set up for mode-3 (or any other mode) operation, the corresponding output pin will be in tristate. As soon as a counter is set up for mode 3, the output becomes logic 1.

The BCounter is loaded from the corresponding counter when any of the following three conditions is satisfied.

- Gate = 1, counter is not presently running, but is loaded;
- Gate makes a 0 to 1 transition;
- Gate = 1, counter is presently running, and TC is reached.

In mode-3 operation, the output will remain high for half the time during the count-down process. For the remaining half-time of count-down process, the output will be in logic 0. If the count value loaded is an odd number, say 4261H, then the output will be in logic 1 state for 2131H clock periods and in logic-0 state for 2130H clock periods. Thus, when the count value is odd, the output will be in logic-1 state for one clock period more than in logic 0 state.

When the TC is reached the BCounter is reloaded from counter value, and the count-down operation continues with the output at logic 1. If the counter is loaded with a value of 4264H and the clock input has a period of 1 μs, then the output waveform in mode 3 will be an infinite series of square waves with 50% duty cycle and a period of 4,264 μs. Thus the output is the same as divide by 4264H of the input clock frequency. Hence mode 3 is called square wave rate generator or simply square wave generator. In mode-3 operation it is possible to have the operation synchronized by software or by hardware.

In software synchronization, the gate input is held at logic 1 throughout. The moment the counter is loaded by the processor, the BCounter is loaded from the counter and the count-down operation begins. The output will be in logic 1 for 50% of the time and logic 0 for the remaining 50% of count-down time. The BCounter is then reloaded from counter value, and the count-down operation continues with the output at logic 1. Later, even if the gate input is made logic 0, the count down continues and generates the last square wave and the operation stops. This is shown in Fig. 25.12 where a count value of 4264H and clock input period of 1 μs are assumed.

In hardware synchronization, the gate input is held at logic 0 initially. Then, even when the processor loads the counter, the BCounter is not loaded from the counter. It is loaded from the counter and the count-down operation begins only when the gate input makes a 0 to 1 transition. The output will be in logic 1 for 50% of the time and logic 0 for the remaining 50% of count down time. The BCounter

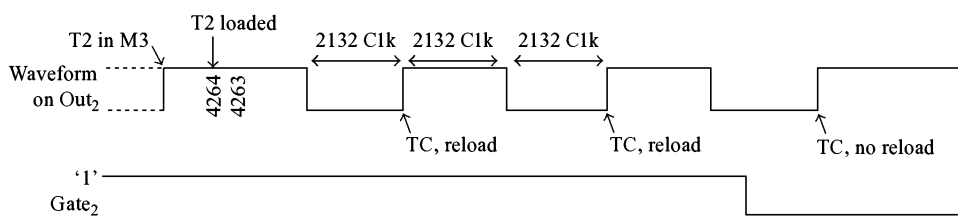


Fig. 25.12 Software synchronization in mode 3

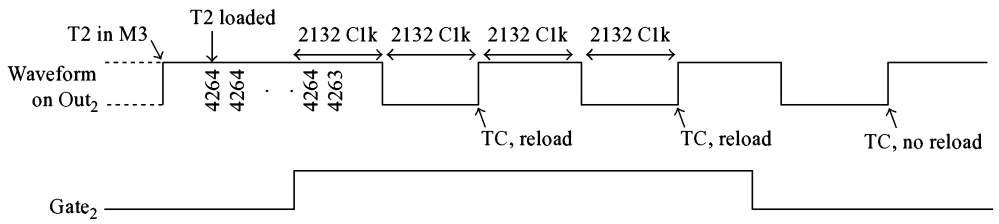


Fig. 25.13 Hardware synchronization in mode 3

is then reloaded from counter value, and the count-down operation continues with the output at logic 1. Later, even if the gate input is made logic 0, the count down continues and generates the last square wave and the operation stops. This is shown in Fig. 25.13 where a count value of 4264H and clock input period of 1 μs are assumed.

Assume that the counter is presently operating in mode 3 with a count value of 4264H. Let us say the counter is loaded with a new value, 5234 when the count down is in progress. This will not affect the count-down process. Thus after 4,264 clock pulses, the TC is reached. The BCounter is then loaded from the counter with the new value of 5,234 and the count down starts again.

Let us say, it is desired to have Counter2 operate in mode 3 with a count value of 3,412 decimal. Then it is necessary to execute the following instructions.

```
MVI A, 10110111B
OUT 83H; Set up Counter2 for Mode 3. Load LS and then MS bytes. Decimal count down

MVI A, 12H
OUT 82H; Load LS byte of Counter2 with 12.

MVI A, 34H
OUT 82H; Load MS byte of Counter2 with 34. Thus load Counter2 with 3412 decimal.
```

■ 25.8 MODE 4—SOFTWARE-TRIGGERED STROBE

Mode 4 is similar to mode 2 except that the operation does not repeat after TC is reached. Before a counter is set up for mode 4 (or any other mode) operation, the corresponding output pin will be in tristate. As soon as a counter is set up for mode 4, the output will become logic 1. The BCounter is loaded from the corresponding counter whenever the processor loads the counter.

In mode-4 operation, the output will remain high during the count-down process. The output becomes logic 0 for one input clock period and then the output becomes logic 1 only for the last count down. Then the BCounter is not reloaded from the counter. If the counter is loaded with a value of 1,234 and the clock input has a period of 1 μs, then the output waveform in mode 4 will be logic-1 level for 1,233 μs and logic 0 for 1 μs, after which the output stays at logic 1. Thus the output is a strobe pulse of one clock period that is generated after 1,233 input clock periods, as shown in Fig. 25.14. Hence mode 4 is called software-triggered strobe.

Let us say, it is desired to have Counter2 operate in mode 4 with a count value of 1,234 decimal. Then it is necessary to execute the following instructions.

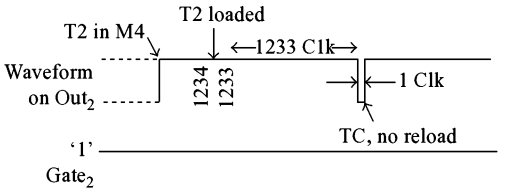


Fig. 25.14
Output waveform in mode 4

MVI A, **10**//1001B
OUT 83H; Set up Counter2 for Mode 4. Load LS and then MS bytes. Decimal count down

MVI A, 34H
OUT 82H; Load LS byte of Counter2 with 34.

MVI A, 12H
OUT 82H; Load MS byte of Counter2 with 12. Thus load Counter2 with 1234 decimal.

If the gate input is made logic 0 during the count down, the count down is inhibited. When the gate input becomes logic 1, the count down continues from where it was stopped. Thus, the generation of strobe pulse is delayed by the time for which the gate input is at logic 0.

This is shown in Fig. 25.15 where a count value of 1,234 and a clock input period of 1 μs are assumed. Let us say the gate input becomes logic 0 when the count down has reached a value of 0800. When the gate input becomes logic 1 again a little later, the count down continues from 0800. Thus, if the $Gate_2$ input is in logic 0 state for 1,000 μs, the strobe signal is generated by Out_2 pin after 2,234 μs.

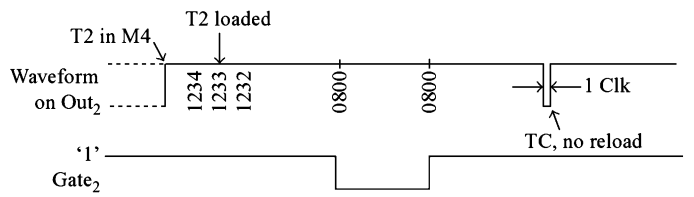


Fig. 25.15
Effect of gate input on counter output waveform in mode 4

If a new value is loaded into the counter while the count down is in progress, it restarts from the new loaded value, thus altering the time at which the strobe pulse is generated.

■ 25.9 MODE 5—HARDWARE-TRIGGERED STROBE

Mode 5 is similar to mode 4 except that the triggering is by hardware instead of software. Before a counter is set up for mode 5 (or any other mode) operation, the corresponding output pin will be in tristate. As soon as a counter is set up for mode 5, the output will become logic 1. The BCounter is loaded from the corresponding counter whenever the corresponding gate input makes a 0 to 1 transition. From then on, the count down of BCounter takes place for every occurrence of clock pulse, immaterial of the logic state of the gate input. In other words, the 1 to 0 transition of the gate input has no effect on the count-down operation.

In mode-5 operation, the output will remain high during the count-down process. The output becomes logic 0 for one input clock period only for the last count down, and then becomes logic 1.

Then the BCounter is not reloaded from the counter. If the counter is loaded with a value of 1,234 and the clock input has a period of 1 μs, then the output waveform in mode 5 will be logic 1 level for 1,233 μs and logic 0 for 1 μs, after which the output stays at logic 1. Thus the output is a strobe pulse of one clock period that is generated after 1,233 input clock periods, after the occurrence of trigger pulse as shown in Fig. 25.16. Hence mode 5 is called hardware-triggered strobe.

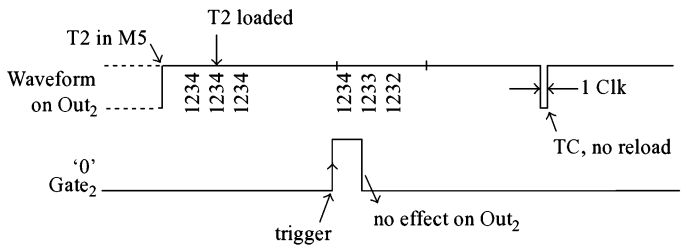


Fig. 25.16
Output waveform in mode 5

Let us say, it is desired to have Counter2 operate in mode 5 with a count value of 1,234 decimal. Then it is necessary to execute the following instructions.

MVI A, **10***11*011**1**B
OUT 83H; Set up Counter2 for Mode 5. Load LS and then MS bytes. Decimal count down

MVI A, 34H
OUT 82H; Load LS byte of Counter2 with 34.

MVI A, 12H
OUT 82H; Load MS byte of Counter2 with 12. Thus load Counter2 with 1234 decimal.

If the gate input makes a 0 to 1 transition during the count-down operation, the count-down operation is restarted from the beginning. This is because the 0 to 1 transition on the gate input causes reloading of the BCounter from the counter. This re-triggering delays the generation of the strobe pulse.

This is shown in Fig. 25.17 where a count value of 1,234 and clock input period of 1 μs are assumed. Let us say the gate input makes a 0 to 1 transition when the count down has reached a value of 0834. Then the count down restarts from 1,234, and so the strobe signal is generated by Out_2 pin after 1,634 μs.

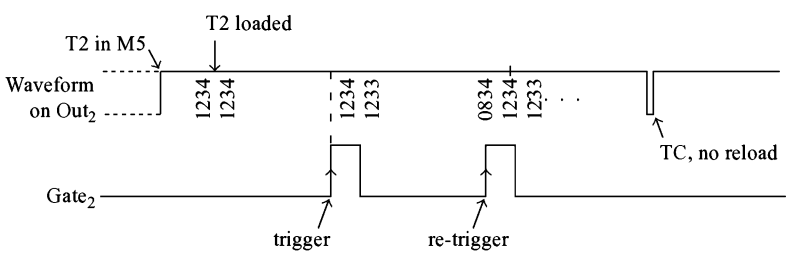


Fig. 25.17 Effect of gate input on counter output waveform in mode 5

If a new value is loaded into the counter while the count down is in progress, it will have no effect on the present output. This value decides the generation of the next strobe output

■ 25.10 USE OF 8253 IN ALS-SDA-85 KIT

Use of 8253 timers in the ALS-85 Kit is as follows:

- The addresses are C8H, C9H, CAH, and CBH for Timer0, Timer1, Timer2, and control port, respectively.
- Timer0 is used in mode 0 (interrupt on TC). It is used for single-step function.
- Timer1 is used in mode 3 (square wave generator). The Clk_1 input is provided with 1.5 MHz. It is derived from the output of a flip-flop that divides the 3-MHz clock-out output of 8085 by 2.
- Timer1 is used to provide the transmit/receive baud rate clock for the 8251 USART. Thus only Timer2 is freely available to the user for his application.

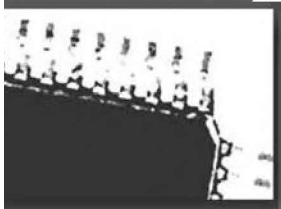
[redacted]

1. Describe the need for a programmable timer in a microcomputer system.
2. Provide a brief overview of the working of 8253 Programmable timer.
3. Briefly describe the functions of the pins of 8253 timer.
4. Describe the function of the registers available in 8253.
5. Provide interpretation of the bits of the Control port in 8253.
6. Write the program segment to configure the timers in 8253 as indicated below. Assume Timer0 address is 60H and Control port address is 63H.

   a. Timer0: Mode 2, counting in decimal, Load 16 bit value
   b. Timer1: Mode 0, counting in hexadecimal, Load LS 8 bit value
   c. Timer2: Mode 4, counting in decimal, Load 16 bit value

7. Explain the need for Read on the fly operation. Describe its implementation in 8253.
8. Explain with neat waveforms Mode 0 operation of 8253. Describe the role of Gate input in this mode.
9. Explain with neat waveforms Mode 1 operation of 8253. Describe the role of Gate input in this mode.
10. Explain with neat waveforms Mode 2 operation of 8253. Describe the role of Gate input in this mode.

11. Explain with neat waveforms Mode 3 operation of 8253. Describe the role of Gate input in this mode.
12. Explain with neat waveforms Mode 4 operation of 8253. Describe the role of Gate input in this mode.
13. Explain with neat waveforms Mode 5 operation of 8253. Describe the role of Gate input in this mode.



26 Intel 8251A—Universal Synchronous Asynchronous Receiver Transmitter (USART)

- Need for a USART
- Asynchronous transmission
- Asynchronous reception
- Synchronous transmission
- Synchronous reception
- Description of 8251 USART
- Programming the 8251
  - *Mode instruction*
  - *Command instruction*
  - *Identifying the command in the control port*
  - *Status port of 8251*
- Use of SOD pin of 8085 for serial transfer
- Questions

This chapter deals in depth with Intel 8251A, which is a universal synchronous asynchronous receiver transmitter (USART). The 8251A is given the name USART for its capability to virtually support any serial data format. This chapter gives a detailed description of the topics mentioned in the chapter outline with neat diagrams.

26.1 NEED FOR USART

A USART is also called a programmable communications interface (PCI). When information is to be sent by 8085 over long distances, it is economical to send it on a single line. In 8085, SOD pin can be used for this purpose, but then the 8085 has to convert 8-bit parallel data to serial data and then output it on SOD pin. Thus lot of CPU time is required for such a conversion. A program to send out serial data on SOD pin is provided at the end of the chapter.

Similarly, if 8085 receives serial data over long distances on SID pin, the 8085 has to internally convert this into parallel data before processing it. Again, lot of time is required for such a conversion.

The 8085 can delegate the job of conversion from serial to parallel and vice versa to the 8251A USART used in the system. The 8251A converts the parallel data received from the processor on the D_{7-0} data pins into serial data, and transmits it on TxD (transmit data) output pin of 8251. Similarly, it converts the serial data received on RxD (receive data) input into parallel data, and the processor reads it using the data pins D_{7-0}.

In this book, 8251A will henceforth be referred to as 8251 for simplicity. The 8251 can support virtually any serial data format, and hence the name 'universal' synchronous asynchronous receiver transmitter. Thus there are specialized I/O port chips that can be assigned some specific tasks, instead of implementing them by software. Implementation by hardware is always much faster too and the processor is free to concentrate on those tasks, which can be taken up by the processor alone. Some of the specialized I/O port chips and their corresponding functions are indicated as follows.

- Intel 8259—Programmable interrupt controller;
- Intel 8257—Programmable DMA controller;
- Intel 8253—Programmable interval timer;
- Intel 8279—Programmable keyboard and display controller;
- Intel 8251—USART.

■ 26.2 ASYNCHRONOUS TRANSMISSION

As the 8251 is used for communication purpose, it is necessary to study the various types of communications. In asynchronous transmission mode, the transmission of characters is not at regular intervals. The transmission is not synchronized with a clock.

The parallel data, to be transmitted in serial format, is sent by the processor to the transmit buffer of 8251. The transmit buffer is an 8-bit port that can only be written, but not read, by the processor. The processor writes to the transmit buffer by activating the CS* input and WR* inputs of 8251, when C/D* (control/data*) input is at 0. The information in the transmit buffer is automatically transferred to transmit shift register. This register acts like a parallel-in serial-out shift register.

When the 8251 chip is selected, the RD*, WR*, and C/D* inputs decide which register is going to be accessed by the processor, as indicated in the following.

*C/D**	*RD**	*WR**	*Action*
0	0	1	Read from receive buffer
0	1	0	Write to transmit buffer
1	0	1	Read status register
1	1	0	Write to control register

For the chip select circuit shown in Fig. 26.1, the 8251 is connected as an I/O-mapped I/O device. The transmit buffer and receive buffer have the same address 50H and the control and status registers have the same address 51H, as per this figure.

Intel 8251 is a programmable chip. It can be configured to suit our requirement by writing to the control port of 8251. The control port is an 8-bit port that can only be written, but not read, by the processor. The processor writes to the control port by activating the CS* and WR* inputs of 8251, when C/D* input is at 1. The control port of 8251 is used to supply the following types of informations to the 8251.

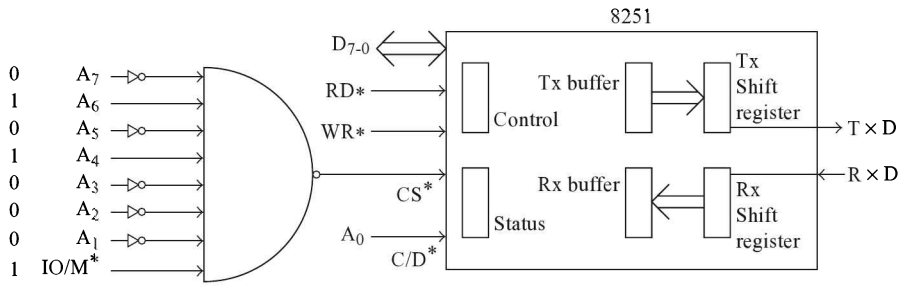


Fig. 26.1 Chip select circuit for connecting 8251 as I/O-mapped I/O

- Mode instruction (MI);
- Synchronization character or characters;
- Command instruction (CI).

Using the MI (to be described later), the 8251 can be configured as indicated in the following list in the asynchronous mode for *transmission as well as reception.*

- 5, 6, 7, or 8 bits per character;
- Even, odd, or no parity;
- 1, 1.5, or 2 stop bits;
- ×1, ×16, or ×64 mode.

Number of Bits per Character: ASCII code is normally used for representing a character. It uses a 7-bit code. Extended ASCII code that uses 8 bits are used when some special characters like graphic characters are to be represented. Intel 8251 allows the user to specify the character length as 5, 6, 7, or 8 bits by writing appropriate information in MI.

Parity Bit: Serial communication is basically used for long distance communication. During transit, the data may get corrupted because of noise on the communication medium. The receiver of the data needs to be sure that he has received the correct data. This can be achieved to a great extent by appending a parity bit at the end of the character data. There are two types of parity bits—Odd parity and Even parity.

The bit appended at the end of the character is called odd parity bit if the number of 1s in the data including the parity bit is an odd number.

Example1: Assume the data to be 01010010. In this, the number of 1s is 3, which is already an odd number. Thus the odd parity to be appended at the end of this character will be a 0. The character with appended odd parity bit will then be 01010010 **0**.

Example 2: Assume the data to be 01110010. In this, the number of 1s is 4, which is an even number. Thus the odd parity to be appended at the end of this character will be a 1, so that there may be odd number of 1s after appending the parity bit. The character with appended odd parity bit will then be 01110010 **1**.

The bit appended at the end of a character is called even parity bit if the number of 1s in the data including the parity bit is an even number.

Example1: Assume the data to be 01010010. In this, the number of 1s is 3, which is an odd number. Thus the even parity to be appended at the end of this character will be a 1, so that there may be even number of 1s after appending the parity bit. The character with appended even parity bit will then be 01010010 **1**.

Example 2: Assume the data to be 01110010. In this, the number of 1s is 4, which is already an even number. Thus the even parity to be appended at the end of this character will be a 0, so that there may be even number of 1s after appending the parity bit. The character with appended even parity bit will then be 01110010 **0**.

Sometimes the data is transferred in serial form over only a small distance. An examle is the downloading of the machine code of 8085 program from a personal computer to the 8085 kit using the serial port of the PC. In such a case, it may not be necessary to append a parity bit at the end of each character. This improves the speed of transmission. Intel 8251 can be programmed for appending an odd parity bit, even parity bit, or not appending any parity bit by writing appropriate information in the MI.

Start and Stop Bits: In the asynchronous mode of transmission, there can be any amount of time gap between the transmissions of two characters. Hence there is a need for the receiver to be informed about the beginning and end of the character. This is done using the start bit that is appended at the beginning of the character and stop bits that are appended at the end of the character.

When there is nothing to transmit, the TxD output of 8251will be in the mark state, which is a logical 1. The receiver then comes to know that the transmitter is active, but has nothing to transmit. Hence the moment the transmitter has a character to send, it sends the *start bit, which is always a logical 0*. This is followed by the data bits of the character, with LS bit transmitted first and MS bit last. The parity bit, as discussed here, is computed by the 8251 and is sent next. Finally, stop bits are sent. The number of stop bits to be transmitted at the end of a character can be programmed to be 1, 1.5, or 2 bits. The *stop bit value is always a logical 1.*

In asynchronous mode, the receiver frequency can be slightly off from the transmitting frequency without causing any problems in receiving. This is because the start bit ensures synchronization at the beginning of every character. If the number of stop bits is programmed for 1.5 or 2 bits there is that much extra time for the receiver to catch up with the transmitter frequency.

Number of Clocks for Transmitting or Receiving a Bit: Intel 8251 uses transmit clock (TxC*) input to send out the information in transmit shift register. For every falling edge of TxC*, a bit of transmit shift register is sent out on TxD output if 8251 is programmed for ×1 mode. In ×16 mode, a bit is sent out for every 16 clock transitions on TxC*. In ×64 mode, a bit is sent out for every 64 clock transitions on TxC*. This is true only for asynchronous operation. *In synchronous mode of operation, a bit is sent out for every falling edge of TxC**.

Example: Let us say, 8251 is configured for asynchronous data transmission with character length of 5 bits, even parity, 1.5 stop bits, and ×1 mode.

Assume that the 8251 transmit buffer is loaded with 35H using the instructions

```
MVI A, 35H
OUT 50H
```

Then the waveform on TxD output pin of 8251 will be as shown in Fig. 26.2. The 8251 TxD output will be in mark state (logic 1) initially. When transmit buffer is loaded with 35H, it is automatically moved to transmit shift register. The contents of transmit shift register are sent out in serial on TxD pin only if CTS* (clear to send) input pin is activated and TxEn (transmitter enable) bit is set to 1 in the CI.

The data is shifted out on the falling edge of TxC* for every 1, 16, or 64 clock pulses depending on whether the 8251 is programmed for ×1, ×16, or ×64 modes respectively.

If conditions for transmission are met, the 8251 sends out the start bit (a logic 0) on TxD pin. Then the data bits of the character are sent out on TxD, starting with the LS bit. In the aforementioned example, only the LS 5 bits of 35H that is **10101** are sent out followed by the even parity bit which is

logic 1 in this case. This is finally followed by 1.5 stop bits that are in the logic 1 state. At the end of all this, the TxD output will again be in the mark state.

It is to be noted that the transmit buffer and transmit shift registers are only 8-bit long. The start bit, calculated parity bit (if any), and stop bits are sent out automatically from internal registers that are not accessible to the user.

The data transmission takes place at the rate of 1 bit per clock cycle, as 8251 is programmed for $\times 1$ mode. If the TxC* frequency is 1 kHz, it needs 8.5 μs for the transmission of the 5-bit character including start bit, parity bit, and the stop bits. The baud rate can be a maximum of 19.2 kHz for asynchronous transmission.

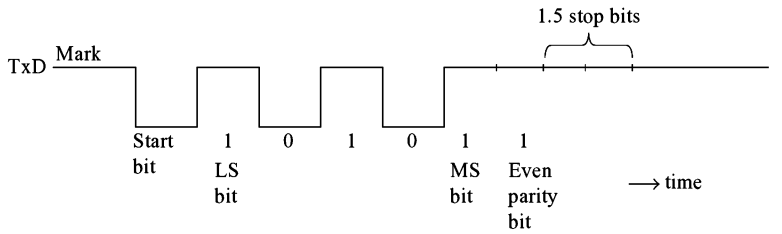


Fig. 26.2 Waveform on TxD in asynchronous transmission

The transmitter is double buffered, which means that when the data in transmit shift register is being shifted out, the transmit buffer is ready to receive another character from the processor. To indicate this situation, the TxRdy (transmitter ready) output of 8251 is activated. To be precise, TxRdy output is activated when the following are true.

- Transmit buffer is empty;
- TxEn bit is set to 1 in the CI;
- CTS* input is active.

The TxRdy output can be used to interrupt the processor. The processor can respond by sending a character to 8251 in the ISS. Assume that the processor is in disable interrupt state, or it does not send a character when it executes the ISS. A short while later the transmit shift register would send out its contents in serial form on TxD pin. Then both the transmit buffer and the transmit shift register are empty. In such a case, the TxE (transmitter empty) output is activated by the 8251.

The TxE output can also be used to interrupt the processor. It is deactivated by the 8251 when a character is written to the transmit buffer by the processor. This output indicates to the processor that the transmitter section has become completely empty and it is time to switch over from transmit mode to receive mode in the half duplex mode of operation.

■ 26.3 ASYNCHRONOUS RECEPTION

In asynchronous receive mode, the receiving of characters is not at regular intervals. The reception is not synchronized with a clock. The 8251 reveives serial data on RxD input pin and stores it in receive shift register, which acts like a serial-in parallel-out shift register. The information in the receive shift register is automatically transferred to receive buffer. The receive buffer is an 8-bit port that can only be read, but not written, by the processor. The processor reads from the receive buffer by activating the CS* and RD* inputs of 8251.

Another important point to be noted is that when the transmitter section of 8251 is configured, the receiver section is also configured with the same features. Thus, as mentioned earlier, the 8251 can be configured using MI in the asynchronous receive/transmit mode for:

- 5, 6, 7, or 8 bits per character;
- Even, odd, or no parity;
- 1, 1.5, or 2 stop bits;
- $\times 1$, $\times 16$, or $\times 64$ mode.

During asynchronous reception, the 8251 receives a character of 5- to 8- bits length on RxD pin, along with start bit (logic 0) at the beginning, parity bit and stop bits (logic 1) at the end. A bit is received by 8251 for every 1, 16, or 64 clock pulses on RxC* input for $\times 1$, $\times 16$, and $\times 64$ modes, respectively.

The RxD pin will normally be in the mark state (logic 1). A $1 \rightarrow 0$ transition on this line is treated as the beginning of the start bit, if the 8251 is programmed for $\times 1$ mode. The RxD pin is sensed at every $0 \rightarrow 1$ transition of RxC* in the $\times 1$ mode.

In the $\times 16$ and $\times 64$ modes there is 'false start bit' detection. A $1 \rightarrow 0$ transition on RxD pin triggers the detection of the start bit, if the 8251 is programmed for $\times 16$ or $\times 64$ mode. If it is $\times 16$ mode, the RxD pin value is again checked at the end of eight RxC* clocks on the rising edge of RxC*. It is treated as a valid start bit only if logic 0 is detected again. From then on, the RxD pin is sensed at the end of every 16 RxC* clock pulses. This ensures that a bit value is sensed at its nominal mid position.

Similarly, in $\times 64$ mode, the RxD pin is again checked at the end of 32 RxC* clocks on the rising edge of RxC*. Only if logic 0 is detected again, it is treated as a valid start bit. From then on, the RxD pin is sensed at the end of every 64 RxC* clock pulses. This ensures that a bit value is sensed at its nominal mid polistion.

If incorrect parity bit is received by the 8251, it sets the parity error (PE) bit in the status register. If logic 0 is detected, when the 8251 is expecting the occurrence of a stop bit, it sets the framing error (FE) bit in the status register. The 8251 is satisfied with receiving one stop bit—it does not matter if it was programmed for 1, 1.5, or 2 stop bits.

Once a character is received in the receive shift register without PE and FE, it is automatically moved to the receive buffer. Then the RxRdy (receiver ready) output pin of 8251 is activated to indicate that there is data in the receive buffer for the processor to receive it. This happens assuming that RxEn (receiver enable) bit is set to 1 in the CI. The RxRdy pin is deactivated when the processor reads the receive buffer.

The receiver is double buffered, which means that when the data in the receive buffer still remains to be read by the processor, the receive shift register is ready to receive another character.

The RxRdy output can be used to interrupt the processor. The processor can respond by reading a character from 8251 in the ISS. Assume that the processor is in disable interrupt state, or it does not read a character when it executes the ISS. A short while later the receive shift register may receive another character and send it to the receive buffer. Then the earlier character in the receive buffer that is yet to be read by the processor is overwritten. This error sets the over-run error (OE) bit in the status register.

Although one or more of the three error bits are set, the 8251 continues with the receive operation without any effect. It is to be noted that the receive buffer and receive shift register are only 8-bit long. The start bit, parity bit (if any), and the stop bits are automatically received in internal registers that are not accessible to the user.

Example: Let us say, the 8251 is configured for asynchronous data reception with character length of 5 bits, even parity, 1.5 stop bits, and $\times 1$ mode.

If the serial input is as shown in Fig. 26.3 on RxD input pin of 8251, then the data shifted into the receive shift register will be 10110, with the MS 3 bits as 000. This is the value that is then transferred to the receive buffer.

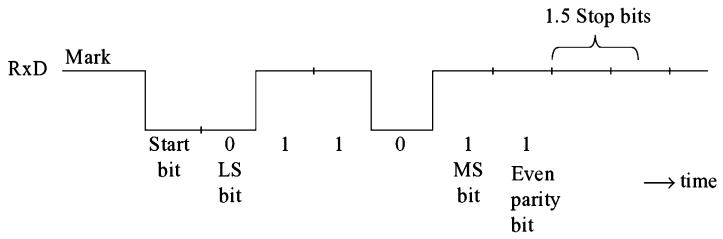


Fig. 26.3
Waveform on RxD in asynchronous reception

If the processor executes `IN 50H` instruction, then the 8251 chip is selected, RD* input is activated, and C/D* input becomes 0. The accumulator contents become 000 10110. The processor now ignores the MS 3 bits, and recognizes the character data as 10110. The RxRdy pin is now deactivated.

■ 26.4 SYNCHRONOUS TRANSMISSION

In synchronous transmission, characters are sent one after another without any gap, synchronized by clock pulses. Before the actual synchronous transmission, the processor writes to 8251 control port one or two synchronization characters, depending on the way the 8251 is configured. Thus the 8251 is aware of the sync (synchronization) characters to be used in the synchronous mode.

First of all, the 8251 sends out the programmed number of sync characters on the TxD pin. This is followed by the assembled data characters. The assembled data characters will not have any start or stop bits. The start and stop bits are not needed any more as one data character follows another without any time gap between them. However, the assembled data characters may have the optional parity bit. Since there are no start and stop bits for each character, the synchronous mode of transmission is faster. It is to be noted that in synchronous mode, a bit is sent out of TxD pin for every TxC* clock. It does not support ×16 and ×64 modes.

The 8251 expects a steady stream of data characters from the processor for transmission on TxD output pin. In case the transmitter becomes empty, indicated by TxE signal becoming active, the 8251 automatically sends out programmed number of sync characters on TxD pin to avoid losing synchronization. The waveform on TxD in synchronous transmission mode is shown in Fig. 26.4.

In synchronous mode, the receiver frequency must match the transmitting frequency. Else, it causes problems in receiving. This is because there is no start bit that ensures synchronization at the beginning of every character.

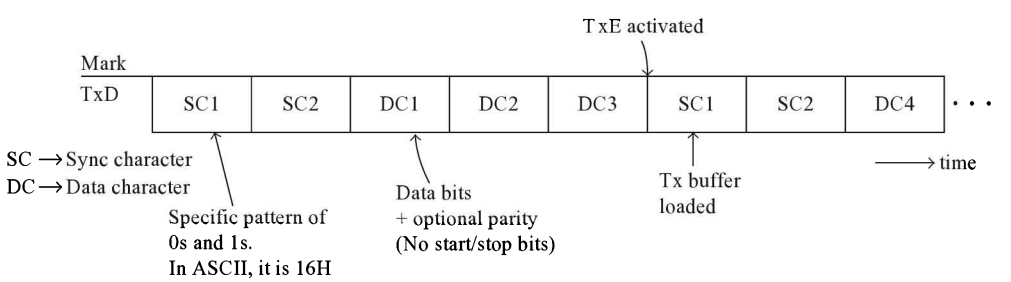


Fig. 26.4 Waveform on TxD in synchronous transmission assuming two sync characters

■ 26.5 SYNCHRONOUS RECEPTION

Waveform on RxD in synchronous receive mode is shown in Fig. 26.5. In synchronous mode, the receiver frequency must match the transmitting frequency. Else, it causes problems in receiving. This is because there is no start bit that ensures synchronization at the beginning of every character.

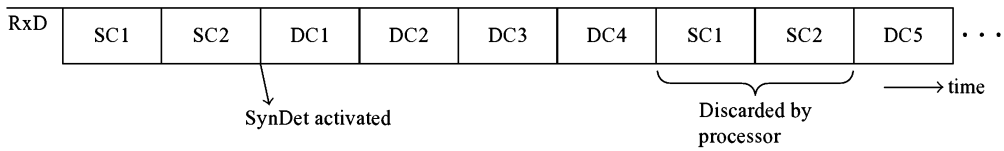


Fig. 26.5 Waveform on RxD in synchronous reception assuming two sync characters

In this mode, character synchronization can be achieved either internally or externally. It depends on bit 6 of the MI, when programmed for synchronous operation. This bit may be called ESD (external sync detect) bit, when programmed for synchronous operation.

If ESD bit = 1, 8251 is prgrammed for external sync detection.
If ESD bit = 0, 8251 is programmed for internal sync detection.

Before any data is received by 8251 in internal sync detect mode, the EH (enter hunt) bit must be set to 1 in the CI. This results in 8251 hunting for programmed number of sync characters on RxD input. The characters received are stored in internal registers that are not accessible to the user. They are compared with the programmed sync characters. Once it detects the sync characters, the 8251 comes out of the hunt mode and activates the SynDet (SD) output. Once the 8251 comes out of the hunt mode it starts receiving characters in the receiver buffer. It is to be noted that in synchronous mode, a bit is received on RxD pin for every RxC* clock. It does not support ×16 and ×64 modes. SD pin is an output pin in internal sync mode. This output is automatically deactivated when the processor reads the status register.

When in external sync mode, the SD pin is an input pin. In this case, external circuit checks for sync characters. Once sync characters are detected, logic 1 is input on the SD input. The 8251 then comes out of the hunt mode and starts receiving characters. The logic 1 on SD pin can be removed after one RxC* cycle.

Once sync characters are detected, the 8251 starts receiving characters in the receive shift register and moves it to the receive buffer for the processor to read. After some data characters are received, if sync characters are again received on the RxD pin, they are received in the receive buffer for the processor to read. It now becomes the responsibility of the processor to identify them as sync characters and discard them, as they do not form part of the data.

■ 26.6 PIN DESCRIPTION OF 8251 USART

Intel 8251 is a 28-pin programmable IC available as a DIP package. Its physical and functional pin diagrams are indicated in Figs. 26.6 and 26.7, respectively.

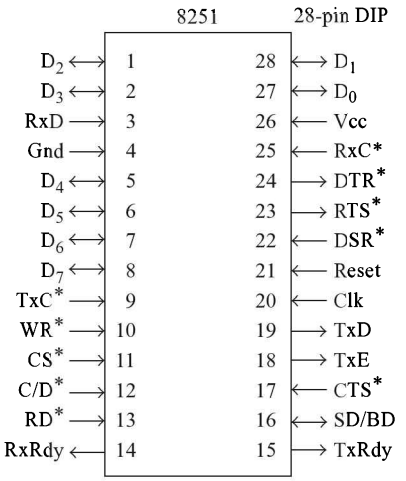


Fig. 26.6
Pin diagram of Intel 8251

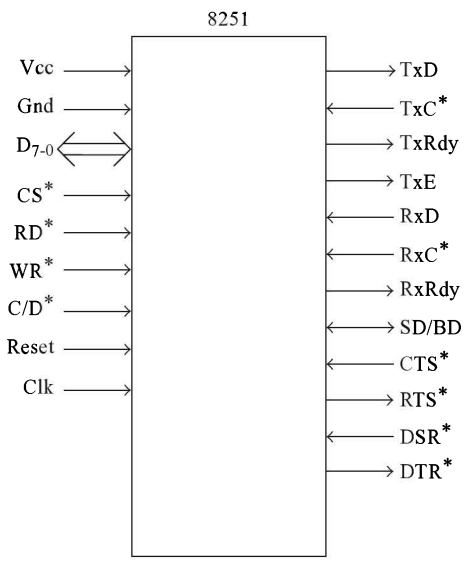


Fig. 26.7
Functional pin diagram of 8251

Vcc and Gnd: Power supply and ground pins. 8251 uses +5V power supply.

D_{7-0}: Eight bi-directional data pins for communication with the processor.

RD*: Active low input pin that is activated by the processor to read status information and receive buffer information from the 8251.

WR*: Active low input pin that is activated by the processor to write to control register and to transmit buffer of the 8251.

CS*: Active low input pin used for selecting the chip.

TxD: It stands for Transmit Data Output pin, on which parallel data received from the processor is sent out in serial fashion.

TxRdy: It is the abbreviation for Transmitter Ready. Active high output pin, which indicates to the processor that the 8251 is ready to receive a character into the transmit buffer. This output can be used to interrupt the processor. The TxRdy output is activated when the following conditions are satisfied.

- Transmit buffer is empty;
- TxEn bit is set to 1 in the CI;
- CTS* input is active.

It is deactivated by the 8251 when a character is written to the transmit buffer by the processor.

TxE: It is the abbreviation for Transmitter Empty. Active high output pin, which indicates to the processor that the transmitter section has become completely empty—both transmit buffer and transmit shift register are now empty. It is time then to switch over from transmit mode to receive mode in the half duplex mode of operation. The TxE output can also be used to interrupt the processor. It is deactivated by the 8251 when a character is written to the transmit buffer by the processor.

TxC*: It is the abbreviation for Transmitter clock. Clock input to the transmit shift register. The data is shifted out of transmit shift register in serial form on the falling edge of TxC*, for every 1, 16, or 64 clock pulses in asynchronous mode. This depends on whether the 8251 is programmed for ×1, ×16, or ×64 modes. In synchronous mode, data is shifted out for every clock pulse.

Clock: The clock input is used to generate internal timing for the 8251. No external inputs or outputs are referenced to the clock input. This frequency must be at least 30 times greater than the transmitter or receiver bit rates.

RxD: It is abbreviation for Receive data. Input pin on which data is received in serial fashion. This is converted to parallel form by the 8251, which is read by the processor.

RxC*: It is abbreviation for Receiver clock. Clock input to the receiver Shift register. The data is shifted into the receiver shift register in serial form on the rising edge of RxC*, for every 1, 16, or 64 clock pulses in asynchronous mode. This depends on whether the 8251 is programmed for ×1, ×16, or ×64 modes. In synchronous mode, data is shifted into the receiver shift register for every clock pulse.

Intel 8251 is normally required to handle the transmission and receive operations of a single link. In such situations, the transmission and reception baud rates are required to be the same. The TxC* and RxC* pins are then connected to the same frequency source.

RxRdy: It is abbreviation for Receiver Ready. Active high output pin, which indicates to the processor that the 8251 has received a character in the receiver buffer. This output can be used to interrupt the processor. The RxRdy output is activated when the following conditions are met.

- RxEn bit is set to 1 in the CI;
- A complete character is received by the receive shift register, and the character (excluding start, parity, and stop bits—in the case of asynchronous mode, and excluding parity bit in synchronous mode) is transferred to the receive buffer.

It is deactivated by the 8251 when a character is read from the receive buffer by the processor.

C/D*: It is abbreviation for Control/Data*. It is an input pin to the 8251. When it is logic 1, the control/status register is selected and when it is logic 0, the data register—transmit buffer/receive buffer is selected. Thus, when the 8251 chip is selected, the RD*, WR*, and C/D* inputs together decide which register is going to be accessed by the processor, as indicated in the following table.

*C/D**	*RD**	*WR**	*Action*
0	0	1	Read from receive buffer
0	1	0	Write to transmit buffer
1	0	1	Read status register
1	1	0	Write to control register

Reset: It is an active high input pin. It should remain in logic 1 state for at least six clock periods of Clk input. It is connected to 'Reset Out' pin of 8085. The 'Reset Out' pin of 8085 is activated when 'Reset In*' input pin of 8085 is activated. Thus, whenever the 8085 is reset, it also resets all the devices connected to 'Reset Out' pin of 8085. After the reset of 8251, it will be in an idle state. It will remain in idle state until a new set of command words are written to the 8251.

SD/BD: It is the abbreviation for SyncDetect/Breakdetect. In the asynchronous mode of operation, this pin is used for break detection. In such a case, it is used as an active high output pin. If the RxD pin remains in logic 0 state for two character durations, the 8251 interprets it as detection of break in receive operation, in asynchronous mode. Then the BD output is activated. This output can be used to interrupt the processor. It is deactivated by the 8251 when the RxD pin receives a logic 1, or the 8251 is reset.

In the synchronous mode of operation, the same pin is used for synchronization detection. The synchronization detection can be internal or external to 8251. This is a programmable feature of 8251. It is decided by bit 6 of MI, when the 8251 is programmed for synchronous operation. If it is external synchronization detection, this pin acts as an input pin to the 8251, informing the 8251 about the detection of synchronization. The logic 1 on this pin can be removed after one RxC* cycle.

If it is internal synchronization detection, this pin acts as an output pin to the 8251. This output can be used to interrupt the processor. It is deactivated by the 8251 when the processor reads the status register of 8251.

Modem control pins: The use of 8251 relieves the processor from the time consuming parallel to serial conversion during transmission, and serial to parallel conversion during receive operation. However, sending information over a long distance on a single line will be expensive, unless the existing cable network, like telephone network, is used for communication. In such a case we have to use audio frequencies. So a MoDem (modulator–demodulator) working as a modulator is used for converting a logic 0 to an audio frequency of 2200 Hz and a logic 1 to another audio frequency of 1200 Hz, and transmitted over a carrier on the telephone network. At the receiving end, another modem, working as a demodulator, converts these frequencies back to 0s and 1s, as shown in Fig. 26.8.

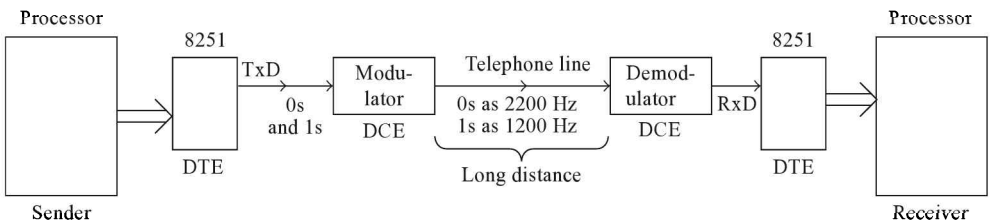


Fig. 26.8 Use of modems for transmission over long distances

Modems are generally called data communication equipment (DCE). The originators and terminators of data are called data terminal equipment (DTE). Thus, 8251 is a DTE. There are instances when serial communication is used even for short distance communication with a peripheral. In such cases, the peripheral is called the DCE.

The interaction between DTE and DCE is handled by the following four modem control signals.

DTR*: It is the abbreviation for Data Terminal Ready. This is an active low output pin. It informs the modem about the readiness of the 8251 to receive or transmit serial data. This pin is activated when the DTR bit in the CI is set to 1. The DTR bit information is inverted and sent out on DTR* pin.

DSR*: It is the abbreviation for Data Set Ready. This is an active low input pin. It is used by the modem to inform the 8251 that it is ready to accept data from the 8251 for transmission. The processor can obtain the logic value on this pin by reading the MS bit of the status register of 8251. The DSR* pin information is inverted and stored in the MS bit of the status register. Thus if the MS bit of status register is 1, it means that the DSR* pin is active.

RTS: It is the abbreviation for Request To Send. This is an active low output pin. It informs the modem that the 8251 has data to be transmitted. This pin is activated when the RTS bit in the CI is set to 1. The RTS bit information is inverted and sent out on RTS* pin.

CTS*: It is the abbreviation for Clear to Send. This is an active low input pin. It is used by the modem to inform the 8251 that the line is clear for the 8251 to send the data. The processor has no means of reading the logic value on this pin, as the status register of 8251 does not provide this information. The information comes out on TxD pin of 8251 only if CTS* is active *and* the TxEn bit in CI is set to 1. When CTS* is deactivated *or* TxEn bit is reset to 0, the transmission is disabled after the transmission of the character in progress is completed.

■ 26.7 PROGRAMMING THE 8251

Prior to data transmission or reception, the 8251 must be loaded with a set of control words. This must follow a reset of 8251. The reset of 8251 can be done in the following two ways.

- By sending a logic 1 pulse on the external reset input pin of 8251. This is hardware reset.
- By setting the IR (internal reset) bit of CI to 1. This is software reset.

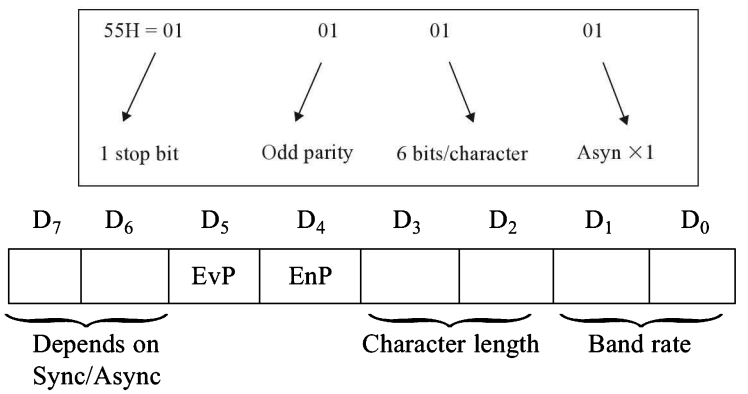
The control port of 8251 is used to supply to the 8251 the following types of information.

- Mode instruction;
- Synchronization character or characters;
- Command instruction.

26.7.1 MODE INSTRUCTION

Mode instruction defines the general operational characteristics of the 8251 as indicated in Fig. 26.9. Once configured, it is not expected to change during the communication process.

Thus, if 55H is the content of control port when it contains MI, it means the following:



D_1, D_0: Configures baud rate factor
00 = Sync mode
01 = Async ×1 mode
10 = Async ×16 mode
11 = Async ×64 mode

D_3, D_2: Configures character length
00 = 5 bits per character
01 = 6 bits per character
10 = 7 bits per character
11 = 8 bits per character

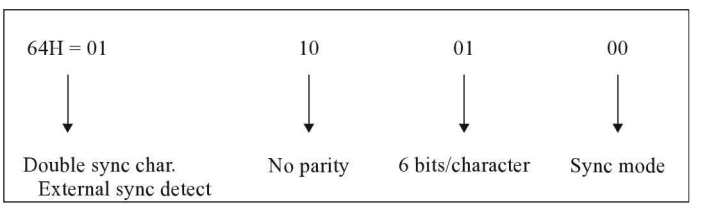
D_5, D_4: Configures parity
D_5 = 1 means even parity (EvP), D_4 = 1 means enable parity (EnP)
00 = No parity
01 = Odd parity
10 = No parity
11 = Even parity

D_7, D_6: Provides different information depending on whether it is asynchronous mode or synchronous mode. Configures the number of stop bits in asynchronous mode (D_1, $D_0 \neq 00$)
00 = Invalid
01 = 1 stop bit
10 = 1.5 stop bits
11 = 2 stop bits

Configures the number of sync characters, external/internal sync detection in sync mode (D_1, D_0 = 00). In this case D_7 = 1 means single sync char, D_6 = 1 means external sync detect
00 = Double sync character, internal sync detect
01 = Double sync character, external sync detect
10 = Single sync character, internal sync detect
11 = Single sync character, external sync detect

Fig. 26.9 Control port containing MI

Similarly, if 64H is the content of the control port when it contains MI, it means the following:



26.7.2 Command Instruction

Command instruction controls the actual operation of the 8251 as indicated in Fig. 26.10. The CI may have to be altered several times during the course of communication. For example, the transmitter might be needed to be disabled, the error bits might have to be reset, and so on.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
EH	IR	RTS	ER	SBRK	RxEn	DTR	TxEn

TxEn: 1 = Transmitter enabled
0 = Transmitter disabled

DTR: 1 = DTR* output pin activated (becomes 0)
0 = DTR* output pin deactivated (becomes 1)

RxEn: 1 = Receiver enabled
0 = Receiver disabled

SBRK: 1 = Send break character on TxD pin
0 = Normal operation

ER: 1 = Reset the PE, OE, and FE bits of status register (ER for 'Error Reset')
0 = Normal operation

RTS: 0 = RTS* output pin activated (becomes 0)
1 = RTS* output pin deactivated (becomes 1)

IR: 1 = Perform internal reset of 8251. This is software reset of 8251.
0 = Normal operation

EH: 1 = Enter hunt mode—Look for sync character on RxD pin, in sync mode only
0 = Normal operation

Fig. 26.10 Control port containing CI

26.7.3 Identifying the Command in the Control Port

The control port contains MI, CI, or sync characters. The 8251 interprets just by the context the contents of the control port as indicated in Fig. 26.11. After a reset operation, the 8251 expects the MI to be written to the control port by the processor. This reset operation could be due to hardware reset using the reset input pin, or software reset using the IR bit of CI. The LS 2 bits of MI clearly indicate whether the 8251 is configured for synchronous or asynchronous mode.

If it is asynchronous mode, the next value written to the control port is interpreted as CI. Any subsequent value written to the control port is interpreted to be a new CI that overrides the previous CI.

If it is in synchronous mode, the MS bit of the MI decides whether the 8251 is configured for one or two sync characters. If it is configured for single sync character the next value written to the control port is interpreted as the sync character. When the control port is written the third time, it is interpreted to be CI. Any subsequent value written to the control port is interpreted to be a new CI that overrides the previous CI.

If it is configured for double sync character the next value written to the control port is interpreted as the first sync character. When the control port is written the third time, it is interpreted to be the second sync character. When the control port is written the fourth time, it is interpreted to be CI. Any subsequent value written to the control port is interpreted to be a new CI that overrides the previous CI.

Thus to enter a new MI, the 8251 has to be reset first before writing to the control port.

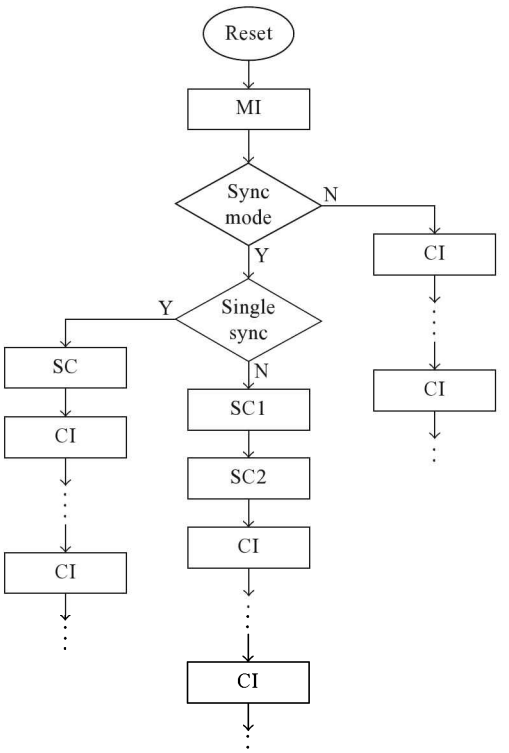


Fig. 26.11
Interpreting the contents of the control port

If the processor writes to the control port five times after the reset of 8251, then the interpretation of control port contents is as follows.

Async mode	*Sync mode—single sync char*	*Sync mode—double sync char*
MI	MI	MI
CI	SC (sync character)	SC1 (sync char 1)
CI	CI	SC2 (sync char 2)
CI	CI	CI
CI	CI	CI

26.7.4 Status Port of 8251

The status information of 8251 is read by the processor when CS* = 0, C/D* = 1, and RD* =0. Both control and status ports have the same address, but control port is written by the processor, while the status port is read by the processor. The contents of the status Port Provide Status information about 8251 as indicated in Fig. 26.12.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
DSR	SD/BD	FE	OE	PE	TxE	RxRdy	TxRdy

TxRdy: 1 = Transmitter buffer is empty
0 = Transmitter buffer is not yet empty

RxRdy: 1 = Receiver has a character for the processor to accept
0 = Receiver does not have a character for the single word to accept

TxE: 1 = Both transmit buffer and transmit shift register are empty—is true
0 = Both transmit buffer and transmit shift register are empty—is false

PE: 1= Parity error occurred during receive operation
0 = No parity error occurrence during receive operation

OE: 1 = Over-run error occurred during receive operation
0 = No over-run error occurrence during receive operation

FE: 1 = Framing error occurred during receive operation
0 = No framing error occurrence during receive operation

SD/BD: 1 = Sync detected/break detected during receive operation
0 = No sync detected/break detected during receive operation

DSR: 1 = DSR* input is active—it is logic 0
0 = DSR* input is inactive—it is logic 1

Fig. 26.12 Contents of the status port of 8251

The RxRdy pin of 8251 need not be connected to any interrupt pin of the processor. The RxRdy bit of status register can be read by the processor to find the status of the RxRdy pin. If this bit is set to 1, the processor can read from the receiver buffer of 8251. This allows status check data transfer as an alternative to interrupt driven data transfer. In a similar way, the TxE and SD/BD bits of the status register can be read by the processor to find out about the status of TxE and SD/BD pins.

The TxRdy status bit only indicates whether the transmit buffer is empy or not. The TxRdy output is activated only when the following are true.

- Transmit buffer is empty;
- The TxEn bit is set to 1 in the CI;
- CTS* input is active.

The PE, OE, and FE error bits on the status register are reset to 0 by setting the ER bit in the CI to 1. However, these error bits do not inhibit the receiver operation, even if they are set to 1.

■ 26.8 USE OF SOD PIN OF 8085 FOR SERIAL TRANSFER

When information has to be sent by 8085 over long distances, it is economical to send it on a single line. The SOD pin can be used for this purpose in 8085. In that case the 8085 has to convert an 8-bit

parallel data to serial data and then output it on SOD pin. Thus a lot of CPU time is required for such a conversion. The program to send out serial data on SOD pin is provided in the following. It is assumed that:

- Register B contains the parallel data to be transmitted in serial form on SOD pin of 8085.
- The baud rate required is 300. Hence the time for sending a bit is 1/300 s=3.33 μs.
- No parity bit used.
- Two stop bits to be used.

Program

```
        MVI A, 01000000B
        SIM ; Send a 0 (Start bit) on SOD pin.
        CALL BITTIME; subroutine for a time delay of 3.33 mSec

        MVI C, 08; C used as counter to transmit 8 data bits
AGAIN:  MOV A, B
        RAR
        MOV B,A;The 3 instructions move the LS bit of B to Cy flag

        MVI A, 10000000B
        RAR; Move a bit of B which is in Cy flag to MS bit of A
        SIM
        CALL BITTIME

        DCR C
        JNZ AGAIN; Come out when 8 data bits sent out on SOD pin

        MVI A, 11000000B
        SIM
        CALL BITTIME    ;  Send a 1 (Stop bit) on SOD pin
        CALL BITTIME    ;  Send a 1 (Second Stop bit) on SOD pin
        HLT
```

If 8251 is available in the system, the user has to just send the parallel data into the transmit buffer of 8251. The rest would be taken care of by the 8251. This justifies the need for 8251 in a microcomputer system.

1. Describe the need for a USART in a microcomputer system.
2. Describe asynchronous data transmission with a neat diagram.
3. Describe asynchronous data reception with a neat diagram.
4. Describe synchronous data transmission with neat diagram.
5. Describe the function of the pins of 8251 USART used in serial transmission.
6. Describe the function of the pins of 8251 USART used in serial reception.
7. Describe the function of the pins of 8251 USART used in modem control.
8. Explain mode instruction of 8251 with an example.

9. How is 8251 configured when the mode instruction value is 73H?
10. Explain the command instruction of 8251 with an example.
11. How is 8251 configured when the command instruction value is 37H?
12. Explain the mechanism in 8251 for identifying the command in the control port.
13. Explain the information available in the status port of 8251.
14. What do we understand about the status of 8251 when the status port contains 65H?
15. Write an 8085 assembly language program to receive serial data using SID pin of 8085. Assume baud rate of 2400, even parity, and 1.5 stop bits.

27

Zilog Z-80 Microprocessor

- Comparison of Intel 8080 with Intel 8085
- Programmer's view of Z-80
- Special features of Z-80
- Addressing modes of Z-80
  - *Relative addressing*
  - *Indexed addressing*
  - *Bit addressing*
- Special instruction types
  - *Rotate and shift instructions*
- Pins of Z-80
- Interrupt structure in Z-80
  - *INT* interrupt*
  - *NMI* interrupt*
- Programming examples
  - *Addition of multi-byte numbers*
  - *Exchange of blocks*
  - *Block movement without overlap*
- Instruction set summary
- Questions

In the previous chapters Intel 8085 microprocessor was dealt with in depth. This chapter and the next deal with two other very popular 8-bit microprocessors. They are Z-80 microprocessor from Zilog Corporation, and M-6800 microprocessor from Motorola Corporation. This will provide the reader with a comparison of the most popular 8-bit microprocessors.

■ 27.1 COMPARISON OF INTEL 8080 WITH INTEL 8085

Intel 8080 is the predecessor of both Intel 8085 and Zilog Z-80. In fact, the main people responsible for the development of Intel 8080 left Intel Company and started their own company by the name Zilog. Thus, a brief study of Intel 8080 will enable the reader to appreciate the improvements that have taken place in Intel 8085 and Zilog Z-80.

The functional pin diagrams of Intel 8080 and Intel 8085 are partly provided in Fig. 27.1. The figure also shows the clock generator chip—Intel 8224 and system controller chip—8228 used with 8080.

The Intel 8224 clock generator provides two-phase clock to the 8080 as shown in Fig. 27.2. The following observations could be made from Fig. 27.1.

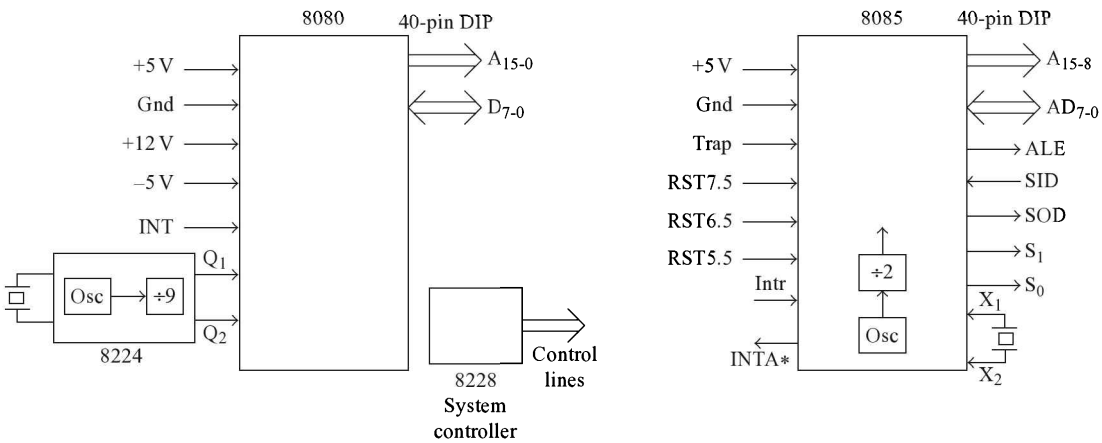


Figure. 27.1 Functional pin diagram of Intel 8080 and Intel 8085

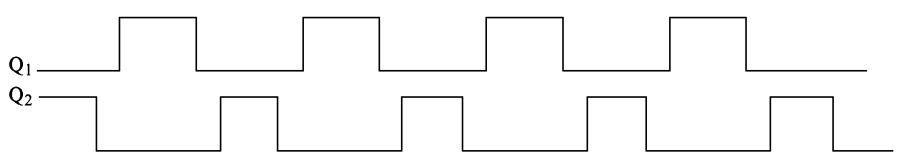


Figure. 27.2 Clock waveform for Intel 8080

Intel 8080	*Intel 8085*
40-pin DIP	40-pin DIP
8 data pins D_{7-0}	8 multiplexed address-data pins AD_{7-0}
16 address pins A_{15-0}	A_{15-8} and AD_{7-0}. ALE pin added
Power supply voltages: +5 V, −5 V, and +12 V	Single power supply of +5 V
Single interrupt pin INT	TRAP, RST7.5, RST6.5, RST5.5 are extra.
No facility for serial communication.	SID and SOD pins, RIM and SIM instructions.
244 opcodes implemented	246 opcodes implemented.
Clock generator chip 8224 needed	Clock generator is internal
Internal clock = Crystal frequency/9	Internal clock = Crystal frequency/2
System controller chip 8228 needed	System controller chip 8228 not needed

Thus the improvements of 8085 over 8080 are detailed as follows.

- Use of single power supply;
- Intel 8080, 8224, 8228 combined into a single Intel 8085;
- Interrupt structure improved;
- Serial communication provided;
- Two instructions—RIM and SIM, added;
- 4.5 times faster than 8080.

27.2 PROGRAMMER'S VIEW OF Z-80

The programmer's view of Z-80 is provided in Fig. 27.3a and the simplified architecture of Z-80 in Fig. 27.3b.

The Z-80 consists of the following registers, the details of which will be dealt with later.

- Set of standard 8-bit registers—A, B, C, D, E, H, L, and F;
- Alternate set of secondary 8-bit registers—A′, B′, C′, D′, E′, H′, L′, and F′;
- Set of 16-bit address registers—PC, SP, IX, and IY;
- Eight-bit I and R registers.

The alternate set of secondary 8-bit registers cannot be used in Z-80 instructions directly. Thus there are no instructions like 'move data from B′ register to D′ register' and 'move data from C register to B′ register'. The only instructions that make use of these alternate set of registers are EXX and EX AF, AF′.

The EXX exchanges B, C, D, E, H, L with B′, C′, D′, E′, H′, L′. It is an 1-byte instruction with the code D9H. It is executed in just four clock cycles. Thus, it is a very fast way of saving B, C, D, E, H, L registers inside the Z-80 itself.

The EX AF, AF′ exchanges AF with A′F′. It is an 1-byte instruction with the code 08H. It is executed in just four clock cycles. Thus, it is a very fast way of saving A and F registers inside the Z-80 itself.

Instruction	*Operation*	*Opcode*
EX AF, AF′	Exchange AF with A′F′	08H
EXX	Exchange B, C, D, E, H, L with B′, C′, D′, E′, H′, L′	D9H

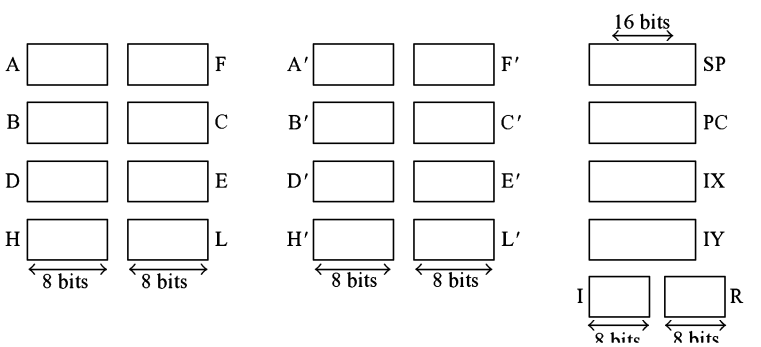


Fig. 27.3a Programmer's view of Z-80

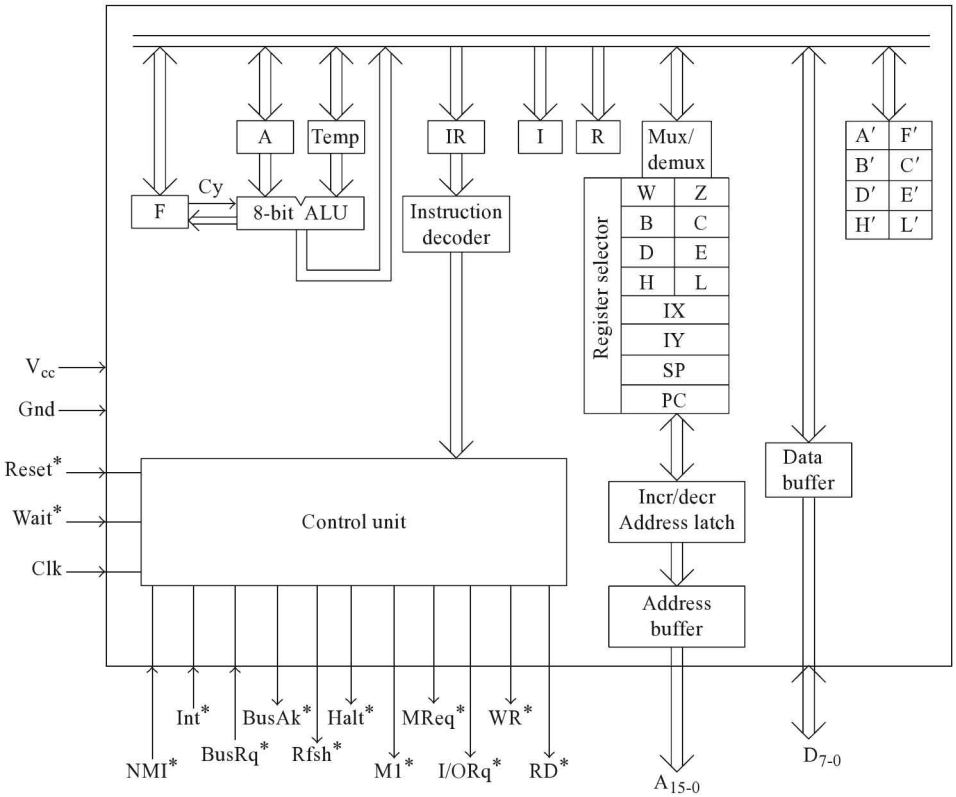


Fig. 27.3b Simplified architecture of Z-80

■ 27.3 SPECIAL FEATURES OF Z-80

Some of the special features of Z-80 are listed in the following.

- 698 opcodes;
- Instruction length: 1 to 4 bytes;
  - 1-byte instructions: 202,
  - 2-byte instructions: 344,
  - 3-byte instructions: 74,
  - 4-byte instructions: 78.
- Opcode length: 1 or 2 bytes;
  - 1-byte opcodes: 252,
  - 2-byte opcodes: 446.
- New addressing modes in comparison with 8085;
  - Indexed addressing,
  - Relative addressing,
  - Bit addressing.

- New instructions in comparison with 8085;
  - Block move,
  - Block search,
  - Powerful I/O instructions,
  - Powerful exchange instructions,
  - Decimal adjustment after addition or subtraction,
  - Instruction to perform program looping,
  - Powerful rotate and shift instructions,
  - Direct memory access type of instructions.
- Powerful interrupt structure.

27.4 ADDRESSING MODES OF Z-80

All the instructions of Intel 8080 are present in Z-80 and many new ones are provided. Only the RIM and SIM of 8085 are not in Z-80. The mnemonic is different in Z-80 and 8080 for the same instruction in a majority of the cases. For example, 'DAD B' instruction of 8080/8085 is the same as 'ADD HL, BC' of Z-80 with the same opcode 09H. Thus the programs written for 8080/8085 can be directly run on Z-80, only after the assembly source code of 8080/8085 is translated to machine language. In the following table, the Z-80 mnemonics and the corresponding 8080/8085 mnemonics are provided. In this table, only the instructions that have different mnemonics in Z-80 and 8085 are included. The convention followed in the table is indicated as follows.

d16 = 16-bit immediate data;
a16 = 16-bit address;
a8 = 8-bit address;
r = A, B, C, D, E, H, or L.

'ADD/ADC A, d8' stands for 'ADD A, d8' and 'ADC A, d8';
'ADD HL, BC/DE/HL/SP' stands for 'ADD HL, BC', 'ADD HL, DE' etc.;
'LD (HL) | r' stands for 'LD (HL), r' and 'LD r, (HL)'.

Z-80 mnemonic	*8080/8085 mnemonic*
ADD/ADC A, d8	ADI/ACI d8
ADD/ADC A, (HL)	ADD/ADC M
ADD/ADC A, r	ADD/ADC r
ADD HL, BC/DE/HL/SP	DAD B/D/H/SP
SUB d8	SUI d8
SUB (HL)	SUB M
SBC A, d8	SBI d8
SBC A, (HL)	SBB M
SBC A, r	SBB r
CALL C/NC/Z/NZ/P/M/PO/PE a16	CNC/CC/CZ/CNZ/CP/CM/CPO/CPE a16
RET C/NC/Z/NZ/P/M/PO/PE	RNC/RC/RZ/RNZ/RP/RM/RPO/RPE
JP a16	JMP a16
JP C/NC/Z/NZ/P/M/PO/PE a16	JNC/JC/JZ/JNZ/JP/JM/JPO/JPE a16
JP (HL)	PCHL
RST 0/8/10H/18H/20H/28H/30H/38H	RST 0/1/2/3/4/5/6/7
INC/DEC r/(HL)	INR/DCR r/M
INC/DEC BC/DE/HL/SP	INX/DCX B/D/H/SP
CCF, SCF	CMC, STC

CP r/(HL)	CMP r/M
CP d8	CPI d8
CPL	CMA
EX DE, HL	XCHG
EX (SP), HL	XTHL
HALT	HLT
IN A, (a8)	IN a8
OUT (a8), A	OUT a8
LD A, (a16)	LDA a16
LD (a16), A	STA a16
LD A, (BC)/(DE)	LDAX B/D
LD (BC)/(DE), A	STAX B/D
LD HL, (a16)	LHLD a16
LD (a16), HL	SHLD a16
LD r1, r2	MOV r1, r2
LD (HL) | r	MOV M | r
LD r/(HL), d8	MVI r/M, d8
LD BC/DE/HL/SP, d16	LXI B/D/H/SP, d16
LD SP, HL	SPHL
PUSH/POP BC/DE/HL/AF	PUSH/POP B/D/H/PSW
RLA/RLCA/RRA/RRCA	RAL/RLC/RAR/RRC

As the Z-80 instruction set is a superset of the 8080 instruction set, it has all the addressing modes of 8080/8085. In addition, there are a number of new addressing modes in Z-80. They are: relative addressing, indexed addressing, and bit addressing. These new addressing modes of Z-80 are described as follows.

27.4.1 Relative Addressing

'JR EXIT' is an example for an instruction that uses relative addressing. Here JR stands for 'jump relative'. It is a 2-byte instruction. EXIT is a symbolic memory location to which the branch occurs when 'JR EXIT' is executed. Assume EXIT is actual memory location 2357H, and 'JR EXIT' instruction is at location 2345H as shown in Fig. 27.4.

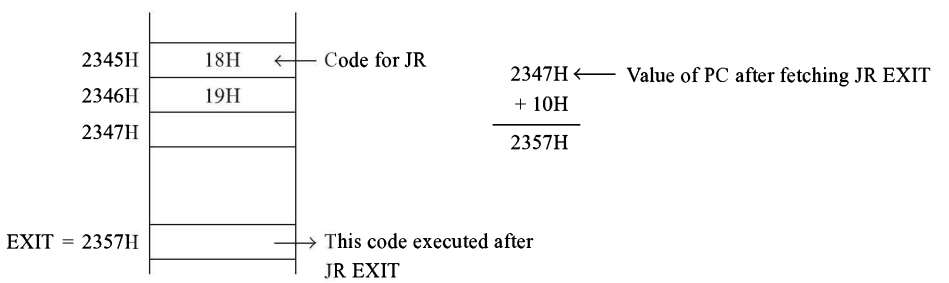


Fig. 27.4 Example for relative addressing

By the time the Z-80 fetches the 'JR EXIT' instruction, the PC would have been incremented to 2347H. Location EXIT is 10H locations further from location 2347H. Thus the assembler generates an 8-bit signed relative displacement value of 10H. Hence the machine code for the instruction would be '18 10'. Here 18H is the opcode for the mnemonic JR, and 10H is the signed displacement corresponding to location EXIT. This 18H happens to be one of the unused opcodes of Intel 8080. The following 12 opcodes were unused in Intel 8080, as only 244 opcodes were implemented: 08H, 10H, 18H, 20H, 28H, 30H, 38H, CBH, D9H, DDH, EDH, and FDH.

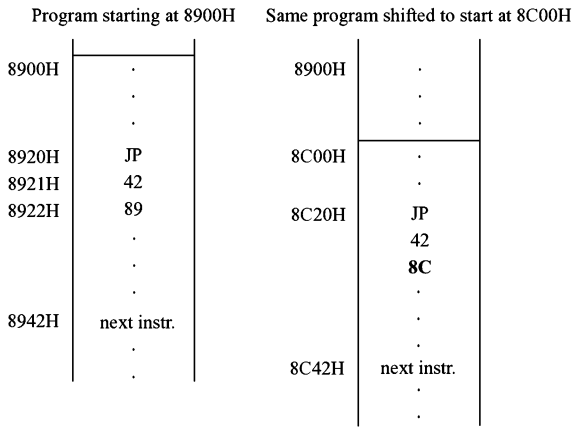


Fig. 27.5a Problem in using absolute jump instruction

If EXIT were to be memory location 2337H, the assembler would have generated an 8-bit signed relative displacement value of F0H, which is nothing but −10H when interpreted as 2's complement signed number. Hence the machine code for the instruction would have been generated as '18 F0'. In general, the JR is coded as '18 r8', where r8 is the 8-bit signed displacement.

Advantages of relative addressing: Let us say we have used absolute jump instructions in our program, as shown in Fig. 27.5a. It may be noted that the JMP instruction of 8085 has the mnemonic JP in Z-80. Thus JP stands for absolute jump in Z-80. Assume that our program is stored in memory starting from 8900H and for some reason, we want to shift our program to memory starting from location 8C00H. If the program is simply moved to a new starting location as 8C00H, it will not work properly in that location. It is necessary to modify the addresses that appear in the absolute jump instructions in order that the program works fine in the relocated region. For example, at location 8C20H the instruction should be changed from 'JP 8942H' to 'JP 8C42H'.

Let us suppose we have used relative jump instructions in our program, as shown in Fig. 27.5b. Assume that our program is stored in memory starting from 8900H and for some reason, we want to shift our program to memory starting from location 8C00H. If the program is simply moved to a new starting location as 8C00H, it will work properly in that location. It is not necessary to modify the

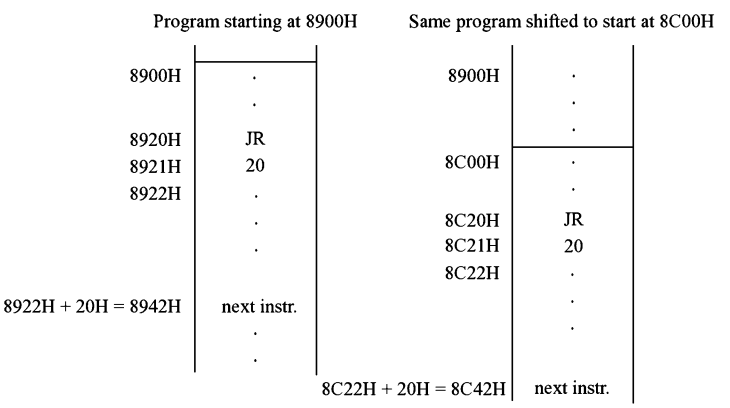


Fig. 27.5b Advantage of using Relative jump instruction

signed displacements that appear in the relative jump instructions in order that the program works fine in the relocated region. For example, at location 8C20H the instruction remains unchanged as 'JR 20H'.

Thus, if relative addressing is used for all branch instructions, the program can be easily relocated to any area of memory. Also, the instruction size is only 2 bytes compared to the 3 bytes required in absolute jump instructions. This reduces the code size and improves the execution speed.

Flags Register F: The flags register contents are indicated as follows. It contains only six flags. The other two bits in the flags register are don't care bits.

S	Z	x	AC	X	P/O	N	Cy

The S, Z, AC, and Cy flags have the same purpose that was assigned to them in 8080/8085. As such, the discussion about their role in Z-80 is omitted. The N flag will be discussed during the explanation of DAA instruction, a little later.

The P/O flag stands for 'parity/overflow' flag. It is a dual-purpose flag, which works as a P flag when the Z-80 executes any of the following instructions.

Rotate and Shift instructions (Ex. RRC D)
AND, OR, XOR instructions

After the execution of these instructions the P/O flag is set to 1 if there are even number of 1's in the result, otherwise it is reset to 0. The P/O flag works as O flag when the Z-80 executes any of the arithmetic instructions (Ex. ADC D).

Overflow flag: The O flag is meaningful only while working with signed numbers. We just ignore this flag with unsigned numbers. Overflow indicates a wrong answer. The processor sometimes gives a wrong answer when it is performing signed number arithmetic. With unsigned number arithmetic, we always get the correct answer. To drive home the aforementioned points, the meaning of O flag is explained as follows with examples.

Addition of unsigned numbers: If we add 42H and 43H. The result will be 85H with Cy flag value as 0, as shown in the following. If we interpret 42H and 43H as unsigned numbers, the answer is correct. In fact, a microprocessor never gives a wrong answer with unsigned arithmetic.

```
  42H
+ 43H
------
  85H with carry = 0
```

Addition of signed numbers: If we add the same two numbers, 42H and 43H, the result will be 85H with Cy flag value as 0, as shown in the following. If we interpret 42H and 43H as signed numbers, the answer is incorrect. A microprocessor sometimes gives a wrong answer with signed arithmetic.

```
  42H  (It is positive, as its MS bit is 0)
+ 43H  (It is positive, as its MS bit is 0)
-------
  85H  (It is negative, as its MS bit is 1, and so overflow set to 1)
```

If we interpret the two numbers as signed numbers, then they are positive numbers, as they start with a 0 in the MS bit position. But the result 85H is negative, as it starts with a 1 in the MS bit position. This is obviously a wrong answer, as the sum of two positive numbers must be positive. Thus, for this example, the O flag will be set to 1.

Let us try to figure out the reason for the occurrence of overflow. The largest positive number using 8 bits is 7FH, and the sum in the example shown is required to be larger than 7FH. So we get a wrong

answer. It may be noted that overflow is a possibility, only when we are adding numbers of the same sign, or subtracting numbers of opposite sign.

It may be noted that

O flag value = C7 ExOR C6

Where, C6 = Cy generated because of the addition performed at bit position 6.
C7 = Cy generated because of the addition performed at bit position 7.
Thus C7 = Cy flag itself.

Other relative jump instructions: There are four conditional relative jump instructions. The jump takes place or not based on a condition indicated by the value of a corresponding flag in the flags register F. The conditional relative jump instructions are shown in Table 27.1.

Table 27.1 Conditional Relative Jump Instructions

Conditional relative jump instruction	*Condition for jumping*	*Opcode*
JR NZ, r8	Z flag = 0	20H
JR Z, r8	Z flag = 1	28H
JR NC, r8	Cy flag = 0	30H
JR C, r8	Cy flag = 1	38H

Notice that there are no relative jump instructions that perform a jump based on the value of S flag and P flag which are not commonly needed. However, if conditional jump based on S flag or P flag is desired, one has to use conditional absolute jump instructions.

Finally, there is one more very useful relative jump instruction which is 'DJNZ r8'. It is also a 2-byte instruction whose opcode is 10H. Its working is as follows.

- It always decrements the contents of B register by 1.
- If the result is non-zero, it performs a backward jump using relative addressing. In DJNZ r8 instructions, r8 is always a negative number.

It is very useful in executing a program segment repetitively for a specified number of times. Thus 'DJNZ BACK' instruction is functionally equivalent to the execution of the following two instructions, which occupy a total of 4 bytes.

```
DEC B
JP NZ BACK
```

Thus six of the 12 unused opcodes of 8080 are used for the six relative jump instructions in Z-80.

Opcode size for Z-80 instructions: As Z-80 has about 700 opcodes, it may seem that opcodes have to be 10-bits long (as $2^9 = 512$ and $2^{10} = 1{,}024$). This problem is ingeniously solved as follows.

The Z-80 instructions that are common with 8080 (a total of 244) will have the same 8-bit opcode as that of 8080. With 8 bits, a maximum of $2^8 = 256$ opcodes are theoretically possible. So another 12 opcodes are possible. These were not used in 8080. Out of them eight opcodes are used in Z-80 to provide eight new instructions. Six of these new instructions come under relative addressing, discussed earlier. The other two are EXX and EX AF, AF′ which are also discussed earlier.

There are still four opcodes remaining which are CBH, DDH, FDH, and EDH. Whenever a Z-80 instruction starts with one of these opcodes it means that it is a pure Z-80 instruction with no parallel in 8080/8085. To exactly find out the new Z-80 instruction, the Z-80 performs second opcode fetch.

Instruction	*Operation*	*Opcode*
EX AF, AF′	Exchange AF with A′F′	08H
EXX	Exchange B, C, D, E, H, L with B′, C′, D′, E′, H′, L′	D9H

The second byte of opcode now indicates the actual new Z-80 instruction. For example, the decoding of 23H by itself, and when preceded by one of the four aforementioned opcodes is shown in Table 27.2.

Table 27.2 Interpretation of Opcode 23H in Z-80

Opcode	*Decoded instruction*
23H	INC HL
DD 23H	INC IX
FD 23H	INC IY
CB 23H	SLA E (Shift Left Arithmetic E register)
ED 23H	Not a valid instruction

With this scheme, $4 \times 256 = 1{,}024$ additional instructions are possible. However only 446 additional instructions are implemented using this scheme.

The new Z-80 instructions that start with the aforementioned four opcodes are briefly indicated in Table 27.3. Instructions that start with the four opcodes as shown in Table 27.3 will have a length of the opcode as 2 bytes.

Table 27.3 Implementation of Pure Z-80 Instructions

First byte of opcode	*Pure Z-80 instructions implemented*	*No. of opcodes*
CBH	Powerful rotate, shift, and bit processing instructions	248
DDH	Instructions that use index register IX	70
FDH	Instructions that use index register IY	70
EDH	Miscellaneous instructions (block move, search, etc.)	58

27.4.2 Indexed Addressing

'ADD A, (IX+20H)' is an example for an instruction that uses indexed addressing. The execution of this instruction results in the addition of A contents with the contents of a memory location. The address of the memory location is partly provided in IX register. The other part of the memory address is provided in the instruction itself. In the aforementioned instruction, the other part of the address is 20H. The 20H in this instruction is commonly called the displacement. It is 8 bits in size, and is an unsigned number. It is denoted as 'd8' in general to indicate displacement of 8 bits size. Indexed addressing is a very powerful addressing mode and is very convenient to access data values in a table.

Example: Assume that A has 30H before the execution of this instruction and the IX content is 8900H. This is interpreted as part of the memory address. The other part of the address provided in the instruction is 20H. Thus the complete memory address is 8920H. If the contents of this location are 35H, then 35H is added to A register contents. The result will be stored in A register. Thus, A contents are changed to 65H after the execution of the instruction. It is indicated diagrammatically as follows.

	Before	*After*
(A)	30H	65H
(IX)	8900H	
(8920H)	35H	

ADD A, (IX+20H) is a 3-byte instruction with the opcode DD 86 20H.
DDH indicates that it is a pure Z-80 instruction that uses IX register.

86H indicates that the instruction is 'ADD A, (IX+d8)'.
20H indicates that the displacement is 20H.

• 27.4.3 Bit Addressing

If the instruction set of Z-80 does not provide bit addressing, then the following five instructions are to be executed for setting bit 3 of B register. In 8085 this method is used for setting bit 3 of B register.

```
PUSH AF;   It is PUSH PSW of 8085. Save A and F on stack top
LD A, B;   It is MOV A, B of 8085. Move B contents to A
OR 08H;    It is ORI 08H of 8085. Set bit 3 of A register
LD B, A;   It is MOV B, A of 8085. Save result in B register
POP AF;    It is POP PSW of 8085. Restore A and F values
```

Bit SET instruction: The Z-80 provides a powerful bit addressing mode. An example for an instruction that uses bit addressing is 'SET 3, B'. The execution of this single instruction results in setting bit 3 of B register to the 1 state. This amply demonstrates the power of bit addressing mode.

'SET 3, B' is a 2-byte instruction with the opcode CB D8H.
CBH indicates that it is a pure Z-80 instruction involving shift, rotate, or bit processing.
D8H = 11 011 000 indicates that the instruction is 'SET 3, B' because of the following.
11 indicates that it is a SET instruction.
N = nnn = 011 indicates that bit 3 is to be set.
R = rrr = 000 indicates that register B is to be used in the instruction.

Similarly, the execution of 'SET 3, (IY+30H)' instruction results in setting bit 3 of a memory location. The address of the memory location is partly provided in IY register. The other part of the memory address is 30H, which is provided in the instruction itself.

If IY content is 8900H, it is interpreted as part of the memory address. The other part of the address provided in the instruction is 30H. Thus the complete memory address is 8930H. Assume that the contents of this location is 35H = 0011 **0**101, with bit 3 value as 0. After the execution of this instruction, the contents of memory location 8930H change to 3DH = 0011 **1**101, indicating that bit 3 has been set to 1.

'SET 3, (IY+30H)' is a 4-byte instruction with the opcode FD CB 30 DEH.
FDH indicates that it is a pure Z-80 instruction involving IY register.
CBH indicates that it is a pure Z-80 instruction involving shift, rotate, or bit processing.
30H provides the displacement value d8 in the indexed addressing mode.
DEH = 11 011 110 indicates that the instruction is 'SET 3, (IY+30H)' because of the following.
11 nnn 110 indicates that it is 'SET N, (Iz+d8)' instruction, where Iz = IX or IY
N = nnn = 011 indicates that bit 3 is to be set.

From these examples, it is clear that there are two general formats for the SET instruction. They are 'SET N, R' and 'SET N, (Iz+d8)', where

N = bit number, a value in the range 0–7
R = A, B, C, D, E, H, L, or (HL) that stands for memory pointed by HL.
Iz = IX or IY
d8 = 8-bit unsigned displacement.

Bit RES instruction: The other bit processing instructions provided in the instruction set of Z-80 are the RES and BIT instructions. RES stands for reset a specified bit. BIT stands for test a specified bit.

The execution of 'RES 4, C' results in resetting bit 4 of register C to the 0 state. It is a 2-byte instruction. Similarly, the execution of 'RES 3, (IY+30H)' instruction results in resetting bit 3 of a memory location. The address of the memory location is partly provided in IY register. The other part of the memory address is 30H. 'RES 3, (IY+30H)' is a 4-byte instruction. The two general formats for the RES instruction are 'RES N, R' and 'RES N, (Iz+d8)'.

BIT instruction: The execution of 'BIT 4, C' results in testing bit 4 of register C. If the bit value is 0, then the Z flag is set to 1, otherwise it is reset to 0. The instruction execution does not alter the bit value. The BIT instruction is generally followed by 'JR NZ r8' or 'JR Z r8' instruction. 'BIT 4, C' is a 2-byte instruction.

Similarly, the execution of 'BIT 3, (IY+30H)' instruction results in testing bit 3 of a memory location. The address of the memory location is partly provided in IY register. The other part of the memory address is 30H. 'BIT 3, (IY+30H)' is a 4-byte instruction. The two general formats for the BIT instruction are 'BIT N, R' and 'BIT N, (Iz+d8)'.

■ 27.5 SPECIAL INSTRUCTION TYPES

DAA Instruction: The DAA instruction of 8085 always adds 00H, 06H, 60H, or 66H depending on the contents of A register and the status of Cy and ACy flags. In other words, it is functionally same as ADI 00H/06H/60H/66H. Intel gave the expansion for 'DAA' instruction of 8085 as 'Decimal Adjust Accumulator'. But a more appropriate expansion would have been '**D**ecimal **A**djust after **A**ddition', because, the DAA instruction of 8085 is supposed to be used for arriving at correct decimal answer only after addition of two 2-digit BCD numbers. It should not be used for arriving at correct decimal answer after subtraction of two 2-digit BCD numbers. In fact, there is no single instruction in 8085 to arrive at correct decimal answer after subtraction of two 2-digit BCD numbers.

As an example, assume that we want to compute 45 − 28. The required answer is 17. If the subtraction were done treating the numbers as hexadecimal, the accumulator contents would be 1DH. Then the execution of DAA instruction of 8085 results in addition of 06H, to get the BCD answer as 23. This is an incorrect result. To get the correct result of 17, 06H would have to be subtracted from 1DH, but there is no single instruction in 8085 to do this.

Now consider Z-80, which has DAA instruction standing for 'Decimal Adjust Accumulator'. It is indeed a very apt expansion in the case of Z-80 because, the DAA instruction of Z-80 can be used for arriving at correct decimal answer immaterial of addition or subtraction of two 2-digit BCD numbers. The DAA instruction of Z-80 is functionally the same as

ADD A 00H/06H/60H/66H if addition was performed prior to execution of DAA
SUB 00H/06H/60H/66H if subtraction was performed prior to execution of DAA

As an example, assume that we want to compute 45 − 28. The required answer is 17. If the subtraction were done treating the numbers as hexadecimal, the accumulator contents would be 1DH. Then the execution of DAA instruction of Z-80 results in subtraction of 06H, to get the correct BCD answer as 17.

But how does Z-80 know whether to add 06H or subtract 06H to arrive at the correct answer in the aforementioned example? This problem is solved by the status of N flag. The value of the N flag is decided by the operation performed by Z-80. It has nothing to do with the result of the operation. Other flags, like Cy, are affected by the result of the operation.

- The N flag is reset to 0 if 8-bit add or increment operation is performed.
- The N flag is set to 1 if 8-bit subtract or decrement operation is performed.

The action taken by Z-80 when DAA instruction is executed depends on the

- value of N flag;
- contents of the accumulator;
- contents of Cy and AC flags.

To summarize, the DAA instruction of Z-80 is functionally the same as

ADD A 00H/06H/60H/66H if N flag = 0
SUB 00H/06H/60H/66H if N flag = 1

LDI Instruction: Assume that we want to move the contents of memory location C200H to memory location C100H. It could be done using the following program segment without altering the contents of register A.

```
LD HL, C200H;   Same as LXI H, C200H of 8085
LD DE, C100H;   Same as LXI D, C100H of 8085
PUSH AF;        Same as PUSH PSW of 8085
LD A, (HL);     Same as MOV A, M of 8085
LD (DE), A;     Same as STAX D of 8085
POP AF;         Same as POP PSW of 8085
```

An easier and more efficient method in Z-80 would be to use the LDI instruction. The execution of this instruction results in the following:

1. Move contents of memory pointed by HL to memory pointed by DE.
2. Increment HL and DE contents by 1.
3. Decrement BC contents by 1.

'LDI' stands for **LoaD** (from memory pointed by HL to memory pointed by DE) and then **I**ncrement HL and DE (BC is always decremented). It is a 2-byte instruction with the opcode ED A0H where:

- EDH indicates that it is a pure Z-80 instruction.
- A0H indicates that it is LDI instruction.

Thus, to move the contents of memory location C200H to memory location C100H, the easier option would be:

```
LD HL, C200H
LD DE, C100H
LDI
```

A more powerful and useful instruction is the LDIR instruction.

LDIR Instruction: Assume that we want to move a block of information of size 6 bytes starting at location C200H to the block at location C100H. If the processor is 8085, we are required to execute a program to achieve this. Such a program was provided in Sect. 15.4 of Chap. 15. A similar program could be written in Z-80 also, but it can be made more simple and efficient using the block move instruction LDIR provided in Z-80.

The execution of the LDIR instruction results in the following:

1. Move contents of memory pointed by HL to memory pointed by DE.
2. Increment HL and DE contents by 1.
3. Decrement BC contents by 1.
4. Repeat steps 1, 2, and 3 if BC content is not 0000H.

'LDIR' stands for **LoaD** (from memory pointed by HL to memory pointed by DE), then **I**ncrement HL and DE (BC is always decremented), and **R**epeat the operation (till BC becomes zero). It is a 2-byte instruction with the opcode ED B0H where:

- EDH indicates that it is a pure Z-80 instruction.
- B0H indicates that it is LDIR instruction.

Thus, to move the contents of a block of 6 bytes of memory starting at location C200H to memory location C100H, the easier option would be:

```
LD HL, C200H
LD DE, C100H
LD BC, 0006H
LDIR
```

LDD Instruction: An alternative method to move the contents of memory location C200H to memory location C100H would be to use the LDD instruction. The execution of this instruction results in the following:

1. Move contents of memory pointed by HL to memory pointed by DE.
2. Decrement HL and DE contents by 1.
3. Decrement BC contents by 1.

'LDD' stands for **LoaD** (from memory pointed by HL to memory pointed by DE) and then **D**ecrement HL and DE (BC is always decremented). It is a 2-byte instruction.

Thus, to move the contents of memory location C200H to memory location C100H, an alternative easier option would be:

```
LD HL, C200H
LD DE, C100H
LDD
```

A more powerful and useful instruction is the LDDR instruction.

LDDR Instruction: Assume that we want to move a block of information of size 6 bytes starting at location C200H to the block at location C100H. An alternative to LDIR instruction is the equally simple and efficient instruction LDDR provided in Z-80.

The execution of the LDDR instruction results in the following:

1. Move contents of memory pointed by HL to memory pointed by DE.
2. Decrement HL and DE contents by 1.
3. Decrement BC contents by 1.
4. Repeat steps 1, 2, and 3 if BC content is not 0000H.

'LDDR' stands for **LoaD** (from memory pointed by HL to memory pointed by DE), then **D**ecrement HL and DE (BC is always decremented), and **R**epeat the operation (till BC becomes zero). It is a 2-byte instruction.

Thus, to move the contents of a block of 6 bytes of memory starting at location C200H to memory location C100H, the equally easy option using LDDR would be:

```
LD HL, C205H
LD DE, C105H
LD BC, 0006H
LDDR
```

CPI Instruction: The execution of CPI instruction results in the following:

1. Compare A register contents with contents of memory pointed by HL.
2. Increment HL contents by 1.
3. Decrement BC contents by 1.

'CPI' stands for **C**om**P**are (register A contents with contents of memory pointed by HL) and then **I**ncrement HL (BC is always decremented). It is a 2-byte instruction with the opcode ED A1H where:

- EDH indicates that it is a pure Z-80 instruction.
- A1H indicates that it is CPI instruction.

A more powerful and useful instruction is the CPIR instruction.

CPIR Instruction: Assume that we want to search for a specific byte value of 23H in a block of information of size 6 bytes starting at location C200H. If the processor is 8085, we are required to execute a program to perform this linear search operation. Such a program is provided in Sect. 16.1 of Chap. 16. A similar program could be written in Z-80 also, but it can be made more simple and efficient using the block search instruction CPIR provided in Z-80.

The execution of the CPIR instruction results in the following:

1. Compare A register contents with contents of memory pointed by HL.
2. Increment HL contents by 1.
3. Decrement BC contents by 1.
4. Repeat steps 1, 2, and 3 if BC content is not 0000H.

'CPIR' stands for **C**om**P**are (A register contents with contents of memory pointed by HL), then **I**ncrement HL (BC is always decremented), and **R**epeat the operation (till BC becomes zero). It is a 2-byte instruction with the opcode ED B1H where:

- EDH indicates that it is a pure Z-80 instruction.
- B1H indicates that it is CPIR instruction.

Thus, to search for a specific byte value of 23H in a block of information of size 6 bytes starting at location C200H, the easier option would be:

```
LD HL, C200H
LD A, 23H
LD BC, 0006H
CPIR
```

The Z flag is set to 1 if the byte value 23H is found anywhere in the block. The location in the block where the byte is found is pointed by HL. If the byte were not found in the block, then the Z flag would be reset to 0. Thus 'JR NZ r8' or 'JR Z r8' instruction normally follows the CPIR instruction.

CPD Instruction: The execution of CPD instruction results in the following:

1. Compare A register contents with contents of memory pointed by HL.
2. Decrement HL contents by 1.
3. Decrement BC contents by 1.

'CPD' stands for **C**om**P**are (A register contents with contents of memory pointed by HL) and then **D**ecrement HL (BC is always decremented). It is a 2-byte instruction. A more powerful and useful instruction is the CPDR instruction.

CPDR Instruction: Assume that we want to search for a specific byte value of 23H in a block of information of size 6 bytes starting at location C200H. For this purpose the CPIR instruction could be used. An equally simple and efficient method is using the block search instruction CPDR in Z-80.

The execution of the CPDR instruction results in the following:

1. Compare A register contents with contents of memory pointed by HL.
2. Decrement HL contents by 1.
3. Decrement BC contents by 1.
4. Repeat steps 1, 2, and 3 if BC content is not 0000H.

'CPDR' stands for **C**om**P**are (A register contents with contents of memory pointed by HL), then **D**ecrement HL (BC is always decremented), and **R**epeat the operation (till BC becomes zero). It is a 2-byte instruction.

Thus, to search for a specific byte value of 23H in a block of information of size 6 bytes starting at location C200H, an equally easier option would be:

```
LD HL, C205H
LD A, 23H
LD BC, 0006H
CPDR
```

'JR NZ r8' or 'JR Z r8' instruction normally follows the CPDR instruction.

IN reg, (C) Instruction: Assume that we want data to be moved from input port number 56H to B register. This could be done using the following program segment.

```
PUSH AF
IN A, (56H);   Same as IN 56H of 8085
LD B, A
POP AF
```

In an 8085-based system, this method was the only way to move data to B register from an I/O port. However, in a Z-80-based system there is a simpler and more efficient method using 'IN reg, (C)' instruction. Here, 'reg' refers to any of the registers A, B, C, D, E, H, or L. For example, the execution of 'IN B, (C)' instruction results in: Move data to register B from input port whose address is provided in C register.

In this type of instruction, C register has a special role to play. Only C register is allowed to have the 8-bit address of the I/O port. Thus to move data from input port number 56H to B register, it is enough to execute the following two instructions.

```
LD C, 56H
IN B, (C)
```

IN B, (C) is a 2-byte instruction with the opcode ED 40H where:
EDH indicates that it is a pure Z-80 instruction.
40H = 01 000 000 indicates that it is 'IN B, (C)' instruction because of the following.
01 rrr 000 indicates that it is 'IN reg, (C)' instruction.
rrr = 000 indicates that register B is to be used in the instruction.

INI Instruction: The execution of INI instruction results in the following:

1. Move data to memory location pointed by HL from input port whose address is provided in C register.
2. Increment HL contents by 1.
3. Decrement B contents by 1.

'INI' stands for **IN**put (from input port whose address is provided in C register to memory pointed by HL) and then **I**ncrement HL (B is always decremented). It is a 2-byte instruction. A more powerful and useful instruction is the INIR instruction.

INIR Instruction: The execution of INIR instruction results in the following:

1. Move data to memory location pointed by HL from input port whose address is provided in C register.
2. Increment HL contents by 1.
3. Decrement B contents by 1.
4. Repeat steps 1, 2, and 3 if B content is not 00H.

'INIR' stands for **IN**put (from input port whose address is provided in C register to memory pointed by HL), then **I**ncrement HL (B is always decremented), and **R**epeat the operation (till B becomes zero). It is a 2-byte instruction, which is used for reading a string of bytes from an input port and storing in consecutive memory locations. It is a DMA type of instruction, as the port directly communicates with memory in this case.

IND Instruction: The execution of IND instruction results in the following:

1. Move data to memory location pointed by HL from input port whose address is provided in C register.
2. Decrement HL contents by 1.
3. Decrement B contents by 1.

'IND' stands for **IN**put (from input port whose address is provided in C register to memory pointed by HL) and then **D**ecrement HL (B is always decremented). It is a 2-byte instruction. A more powerful and useful instruction is the INDR instruction.

INDR Instruction: The execution of INDR instruction results in the following:

1. Move data to memory location pointed by HL from input port whose address is provided in C register.
2. Decrement HL contents by 1.
3. Decrement B contents by 1.
4. Repeat steps 1, 2, and 3 if B content is not 00H.

'INDR' stands for **IN**put (from input port whose address is provided in C register to memory pointed by HL), then **D**ecrement HL (B is always decremented), and **R**epeat the operation (till B becomes zero). It is a 2-byte instruction, which is used for reading a string of bytes from an input port and storing in consecutive memory locations. It is a DMA type of instruction, as the port directly communicates with memory in this case.

OUT (C), reg Instruction: Assume that we want data to be moved from B register to output port number 56H. This could be done using the following program segment.

```
PUSH AF
LD A, B
OUT (56H), A ;   Same as OUT 56H of 8085
POP AF
```

In an 8085-based system, this method was the only way to move data from B register to an I/O port. However, in a Z-80-based system there is a simpler and more efficient method using 'OUT (C), reg'

instruction. Here, 'reg' refers to any of the registers A, B, C, D, E, H, or L. For example, the execution of 'OUT (C), B' instruction results in the following:

- Move data from B register to output port whose address is provided in C register.
- Only C register is allowed to have the 8-bit address of the I/O port.

Thus to move data from B register to output port number 56H, it is enough to execute the following two instructions.

```
LD C, 56H
OUT (C), B
```

'OUT (C), B' is a 2-byte instruction.

OUTI Instruction: The execution of OUTI instruction results in the following:

1. Move data from memory location pointed by HL to output port whose address is provided in C register.
2. Increment HL contents by 1.
3. Decrement B contents by 1.

'OUTI' stands for **OUT**put (from memory pointed by HL to output port whose address is provided in C register) and then **I**ncrement HL (B is always decremented). It is a 2-byte instruction. A more powerful and useful instruction is the OTIR instruction.

OTIR Instruction: The execution of OTIR instruction results in the following:

1. Move data from memory location pointed by HL to output port whose address is provided in C register.
2. Increment HL contents by 1.
3. Decrement B contents by 1.
4. Repeat steps 1, 2, and 3 if B content is not 00H.

'OTIR' stands for **O**u**T**put (from memory pointed by HL to output port whose address is provided in C register), then **I**ncrement HL (B is always decremented), and **R**epeat the operation (till B becomes zero). It is a 2-byte instruction, which is used for sending a string of bytes from consecutive memory locations to an output port. It is a DMA type of instruction, as memory communicates directly with the port in this case.

OUTD Instruction: The execution of OUTD instruction results in the following:

1. Move data from memory location pointed by HL to output port whose address is provided in C register.
2. Decrement HL contents by 1.
3. Decrement B contents by 1.

'OUTD' stands for **OUT**put (from memory pointed by HL to output port whose address is provided in C register) and then **D**ecrement HL (B is always decremented). It is a 2-byte instruction. A more powerful and useful instruction is the OTDR instruction.

OTDR Instruction: The execution of OTDR instruction results in the following:

1. Move data from memory location pointed by HL to output port whose address is provided in C register.

2. Decrement HL contents by 1.
3. Decrement B contents by 1.
4. Repeat steps 1, 2, and 3 if B content is not 00H.

'OTDR' stands for **OuTput** (from memory pointed by HL to output port whose address is provided in C register), then **D**ecrement HL (B is always decremented), and **R**epeat the operation (till B becomes zero). It is a 2-byte instruction.

This instruction is used for sending a string of bytes from consecutive memory locations to an output port. It is a DMA type of instruction, as memory communicates directly with the port in this case.

27.5.1 Rotate and Shift Instructions

Rotate instructions: The rotate instructions of Z-80 that were available in 8085 also are shown in Table 27.4.

Table 27.4 Rotate Instructions of Z-80 that were present in 8085

Z-80 instruction	*Equivalent 8085 instruction*	*Description*
RLCA	RLC	8-bit left rotation
RRCA	RRC	8-bit right rotation
RLA	RAL	9-bit left rotation
RRA	RAR	9-bit right rotation

In 8085 the rotate operations are possible only on A register contents. In Z-80 the rotate operations could be performed on the contents of

- any general purpose 8-bit register or memory location pointed by HL;
- memory location whose address is provided using indexed addressing.

These new rotate instructions of Z-80 are shown in Table 27.5. The following convention is used in this table.

R = A, B, C, D, E, H, L, or (HL);
Iz = IX or IY;
d8 = 8-bit unsigned displacement.

Table 27.5 Pure Z-80 Rotate Instructions

Instruction	*Description*	*Instruction length (bytes)*
RLC R	8-bit left rotation	2
RLC (Iz+d8)	8-bit left rotation	4
RRC R	8-bit right rotation	2
RRC (Iz+d8)	8-bit right rotation	4
RL R	9-bit left rotation	2
RL (Iz+d8)	9-bit left rotation	4
RR R	9-bit right rotation	2
RR (Iz+d8)	9-bit right rotation	4

RLD and RRD Instructions: In addition to the powerful rotate instructions described, there are two more instructions under the rotate group of instructions. They are RLD and RRD.

RLD instruction: RLD stands for 'rotate left digit'. It is a 2-byte instruction. The first byte indicates that it is a pure Z-80 instruction, and the second byte indicates that it is RLD instruction. It works on the contents of A register and the contents of memory pointed by HL. There are a total of four hex digits in A register and the memory contents pointed by HL. However, rotate left operation is performed only on three hex digits, leaving the MS hex digit in register A unchanged. It is shown diagrammatically in Fig. 27.6.

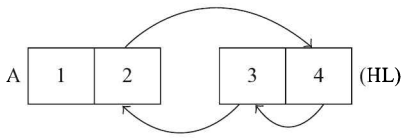


Fig. 27.6
Illustration of RLD instruction

Example:

Before	*After*
A = 12H (HL) = 34H	A = 13H (HL) = 42H

RRD instruction: RRD stands for 'rotate right digit'. It is a 2-byte instruction. The first byte indicates that it is a pure Z-80 instruction, and the second one indicates that it is RRD instruction. It works on the contents of A register and the contents of memory pointed by HL. There are a total of four hex digits in A register and the memory contents pointed by HL. However, rotate right operation is performed only on three hex digits, leaving the MS hex digit in A register unchanged. It is shown diagrammatically in Fig. 27.7.

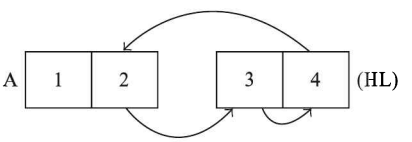


Fig. 27.7
Illustration of RRD instruction

Example

Before	*After*
A = 12H (HL) = 34H	A = 14H (HL) = 23H

Shift instructions: Intel 8085 did not have shift instructions. These are the additions in Z-80. A shift instruction just shifts the number in a register or a memory location to the left or to the right by one bit position. There will not be any rotation of the bits in this case. In Z-80 the shift operations could be performed on the contents of

- any general purpose 8-bit register or memory location pointed by HL;
- memory location whose address is provided using indexed addressing.

The shift instructions are of two types: arithmetic shift and logical shift.

Arithmetic Shift Instructions: Arithmetic shift instructions are used when the number to be shifted is interpreted by the programmer to be a signed number. In arithmetic shift operation the sign of the number remains unchanged after the operation.

SRA instruction: SRA stands for 'shift right arithmetic'. It shifts all the bits to the right by one position. The LS bit that comes out is moved to Cy flag. In the vacancy created at the MS bit position, the earlier MS bit value is retained. The effect of the execution of SRA instruction is indicated in Fig. 27.8.

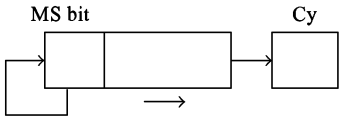


Fig. 27.8
Effect of execution of SRA instruction

A right shift operation divides the signed number by 2, if it is an even number. When an odd number is right shifted, the result is truncated to the nearest integer. For example, when SRA instruction operates on number 5, the result will be 2. Similarly, when SRA instruction operates on number FBH (interpreted as −5), the result will be FDH (interpreted as −3).

The two general formats for the SRA instructions are:

- SRA R of size 2 bytes;
- SRA (Iz+d8) of size 4 bytes

  where

  R = A, B, C, D, E, H, L, or (HL)

  Iz = IX or IY

  d8 = 8-bit unsigned displacement.

SLA instruction: SLA stands for 'shift left arithmetic'. It shifts all the bits to the left by one position. The MS bit that comes out is moved to Cy flag. In the vacancy created at the LS bit position, a 0 value is moved. The effect of the execution of SLA instruction is indicated in Fig. 27.9.

Fig. 27.9
Effect of execution of SLA instruction

The left shift operation has the same effect immaterial of the number to be shifted being treated as signed or unsigned. The condition for correct multiplication by 2 depends on whether the number before shift is treated as unsigned or signed, as shown in the following table with an example each.

Type of number	*Condition for multiplication by 2*	*Before*	*After*
Unsigned	MS bit = 0	40H	80H
Signed	MS 2 bits are same	FEH(=−02)	FCH (=−04)

The two general formats for the SLA instructions are:

- SLA R of size 2 bytes;
- SLA (Iz+d8) of size 4 bytes.

Logical Shift Instructions: Logical shift instructions are used when the number to be shifted is interpreted by the programmer to be an unsigned number.

SRL instruction: SRL stands for 'shift right logical'. It shifts all the bits to the right by one position. The LS bit that comes out is moved to Cy flag. A 0 value is moved in the vacancy created at the MS bit position. The effect of the execution of SRL instruction is indicated in Fig. 27.10.

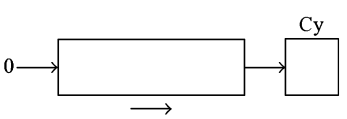


Fig. 27.10
Effect of the execution of SRL instruction

A right shift operation divides the signed number by 2, if it is an even number. When an odd number is right shifted the result is truncated to the nearest integer. For example, when SRL instruction operates on number 5, the result will be 2. Similarly, when SRL instruction operates on number FBH, the result will be 7DH.

The two general formats for the SRL instructions are:

- SRL R of size 2 bytes;
- SRL (Iz+d8) of size 4 bytes.

Similarly, the 'shift left logical' instruction shifts all the bits to the left by one position. The MS bit that comes out is moved to Cy flag. A 0 value is moved in the vacancy created at the LS bit position. This same action is performed when SLA instruction is executed. As such, there is no 'shift left logical' instruction in the instruction set of Z-80. Whenever 'shift left logical' operation is desired, an SLA instruction is executed.

Other Instructions: The remaining pure Z-80 instructions are provided in Table 27.6. The following convention is used in describing the instructions.

Iz = IX or IY;
zp = BC, DE, Iz, or SP (note that HL is not allowed);
d16 = 16-bit immediate data;
a16 = 16-bit address;
a8 = 8-bit address;
r = A, B, C, D, E, H, or L;
rp = BC, DE, HL, or SP.

INC/DEC Iz stands for INC Iz and DEC Iz;
LD Iz | (a16) stands for LD Iz, (a16) and LD (a16), Iz;
IM 0/1/2 stands for IM 0, IM 1, and IM 2.

Table 27.6 Pure Z-80 Instructions with Examples

Instruction	*Typical example*	*Size in bytes*	*No. of opcodes*
ADD Iz, zp	ADD IX, BC	2	8
LD Iz, d16	LD IY, 1234H	4	2
LD (a16) | Iz	LD (ABCDH), IX	4	2 * 2 = 4
INC/DEC Iz	INC IX	2	2 * 2 = 4
INC/DEC (Iz + a8)	INC (IX + 35H)	3	2 * 2 = 4
LD (Iz + a8), d8	LD (IX + 20H), 67H	4	2
LD r | (Iz + a8)	LD B, (IX + 45H)	3	2 * 7 * 2 = 28
PUSH/POP Iz	POP IY	2	2 * 2 = 4
LD SP, Iz	LD SP, IY	2	2
EX (SP), Iz	EX (SP), IX	2	2
JP (Iz)	JP (IX)	2	2
ADD/ADC/SUB/SBC (Iz + a8)	SBC (IX + 23H)	3	4 * 2 = 8
AND/OR/XOR/CP (Iz + a8)	CP (IX + 23H)	3	4 * 2 = 8
ADC/SBC HL, rp	ADC HL, DE	2	2 * 4 = 8
LD (a16) | rp	LD (3456H), BC	4	2 * 4 = 8
NEG	NEG	2	1
RETI	RETI	2	1
RETN	RETN	2	1
IM 0/1/2	IM 2	2	3
LD A | I/R	LD R, A	2	2 * 2 = 4

■ 27.6 PINS OF Z-80

The devices normally used along with Z-80 to provide the required functions in a microcomputer are: Z-80 PIO (similar to 8255 PPI) and Z-80 CTC (similar to 8253 timer).

The actual pin diagram and the functional pin diagram of Z-80 are provided in Figs. 27.11 and 27.12, respectively.

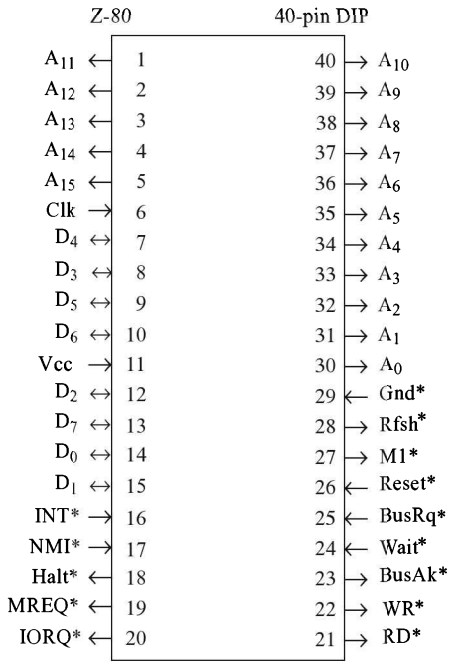


Fig. 27.11
Pin diagram of Z-80

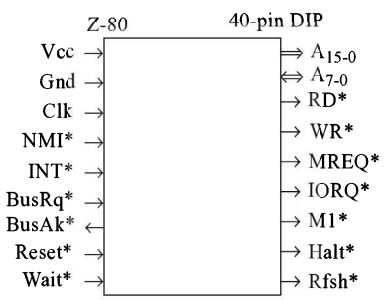


Fig. 27.12
Functional pin diagram of Z-80

$A_{15\text{-}0}$: Unidirectional 16 address pins on which address is sent out by Z-80. Thus, Z-80 is capable of addressing $2^{16} = 64K$ memory locations.

$D_{7\text{-}0}$: Bi-directional 8 data pins on which data is sent out or received by Z-80. Thus, Z-80 is an 8-bit microprocessor.

V_{cc}, Gnd: Power supply pins. They use a dc power supply of +5 V.

Clk: The Z-80 receives the clock input on this pin. An external clock generator generates the clock.

MREQ*: Active low output pin. The Z-80 sends out a 0 on this pin when communication with memory is desired.

IORQ*: Active low output pin. The Z-80 sends out a 0 on this pin when communication with I/O ports is desired.

RD*: Active low output pin. The Z-80 sends out a 0 on this pin when a read operation from memory or an input port is desired.

WR*: Active low output pin. The Z-80 sends out a 0 on this pin when a write operation to memory or an output port is desired.

Reset*: Active low input pin. When momentarily brought down to 0 state, the Z-80 is reset.

Wait*: Active low input pin. It is used to insert wait states when working with slow memory chips.

BusRq*, BusAk*: BusRq* is an active low input pin, which is similar in function to HOLD input of 8085. BusAk* is an active low output pin. It is similar in function to Hlda output of 8085. These two pins are used for facilitating DMA.

NMI*, INT*: These are active low interrupt input pins. NMI stands for non-maskable interrupt and INT stands for interrupt that could be masked or disabled. More details about these interrupts are provided little later.

Halt*: Active low output pin. It is activated by the Z-80 when the HALT instruction is executed.

M1*: Active low output pin. Here M1 stands for Machine cycle number1. In other words, the Z-80 activates this output whenever the first machine cycle of an instruction (opcode fetch machine cycle) is in progress.

Rfsh*: Active low output pin. Here RFSH stands for refresh. This pin is used for refreshing dynamic RAM chips, if any are used in the system. RAM is available in two types: static RAM (SRAM) and dynamic RAM (DRAM). In SRAM, the chip information is stored in flip-flops. It requires about six transistors to implement a flip-flop on the average in a memory chip. The advantage of SRAM is that the information remains intact as long as the power is on. In a DRAM, information is stored on a tiny capacitor as a charge (1 state) or no charge (0 state). It needs only one transistor to write a bit to the DRAM cell or read the bit on the DRAM cell. Thus DRAMs require much less power per bit of storage. Hence more number of bits can be stored in a given size of chip compared to SRAM, which results in lower storage cost per bit. The disadvantage of DRAM is that the charge on the tiny capacitor tends to change due to leakage. Thus, periodic refreshing of information must be done to avoid losing the information. This refresh cycle is essentially a memory access operation and must be performed in a time less than about 2 μs for each of the rows of the DRAM. The R register (refresh register) of Z-80 eases the burden (as briefly described in the following) of dealing with this constant refresh requirement associated with DRAMs.

The contents of R register is incremented automatically by 1 after each opcode fetch. As soon as the opcode byte enters the instruction register, decoding of the instruction starts. This is done in the fourth clock cycle of the opcode fetch machine cycle. The decoding operation is entirely internal to the processor. During this period, the Z-80 outputs the contents of the R register on the LS byte of address bus. Simultaneously, the Z-80 activates Rfsh* output. This results in the refreshing of the entire row of DRAM specified by R register. A few microseconds later, when the next opcode byte is being decoded, the new incremented value of R register and Rfsh* are output. This refreshes the next row of DRAM memory. A typical DRAM could be totally refreshed using this method in about 300 μs, which is well within the limit of 2 μs.

The Z-80 is probably the only 8-bit microprocessor to provide refresh facility on chip for DRAMs. However, it is not normally made use of. This is because of the fact that, SRAMs are used mostly in Z-80-based systems, as the maximum memory addressing capacity is a meager 64K bytes.

■ 27.7 INTERRUPT STRUCTURE IN Z-80

The Z-80 has two external interrupt pins, which are NMI* and INT*. Although there are only two interrupt pins, compared with the five in 8085, the Z-80 has a superior interrupt structure than the 8085. The following description illustrates the superiority of Z-80 interrupt structure compared to 8085.

27.7.1 INT* INTERRUPT

INT* is an active low, level-triggered interrupt input pin. This is an interrupt that could be masked or disabled. The execution of DI instruction disables this interrupt. In such a case, even if an I/O port activates the INT* pin, the Z-80 will not be interrupted. Even after a reset of Z-80 the INT* interrupt is disabled. The Z-80 is always started with activation of reset. Thus, in a Z-80 program, if it is desired that the Z-80 should be interrupted by INT* pin, then the EI instruction must be present in the program. The INT* interrupt is enabled only after the execution of the EI instruction.

The action performed by the Z-80 when INT* pin is activated depends on the interrupt mode of Z-80. There are three interrupt modes that are: interrupt mode 0, interrupt mode 1, and interrupt mode 2.

Interrupt mode 0 (IM 0): This is actually a 2-byte instruction of Z-80 with the opcode ED 46H. After the execution of this instruction the INT* input of Z-80 works in IM 0. In fact, this is the default mode of operation after reset of Z-80. Thus if the interrupt mode type is not explicitly specified by executing one of IM 0, IM 1, or IM 2 instructions, the Z-80 works in IM 0.

INT* in IM 0 mode of operation is very similar to INTR interrupt of 8085. INT* in IM 0 mode of operation acts like a non-vectored interrupt. In this mode the Z-80 will respond to the interrupt request by activating IORQ* and M1* signals. The simultaneous activation of IORQ* and M1* is the Z-80 way of informing the peripherals that it is issuing an interrupt acknowledgement. The Z-80 expects the interrupting peripheral to respond (as IORQ* = 0) with an opcode byte (as M1* = 0). The peripheral should send to the Z-80 an RST instruction or a 3-byte CALL instruction using the data lines of Z-80. The Z-80 will automatically disable the interrupt system. The user is not required to explicitly use DI instruction. The Z-80 will save the address of the next instruction (same as PC contents) above the stack top. Then the Z-80 branches to the ISS indicated by the peripheral using RST or CALL instruction.

Thus the action taken by Z-80 when interrupted half way through an instruction because of INT* in IM 0 mode could be summarized as follows. It is assumed that NMI* is not active and the interrupt system is enabled using EI instruction.

- Completes the execution of the instruction under progress.
- Activates IORQ* and M1*.
- Receives an RST or a CALL instruction from the peripheral.
- Disables the interrupt system.
- Saves PC value (address of next instruction) on the stack top.
- Branches to the ISS indicated by the peripheral.

The format of the INT* service routine will generally be as follows.

- Save all the registers.
- Perform the action needed to satisfy the interrupting device.
- Restore all the register contents.
- Enable the interrupt system using EI instruction.
- Return to main program by executing RETI instruction.

RETI stands for '**RET**urn from **I**NT service routine'. It is a 2-byte instruction with the code ED 4DH. The RETI instruction pops the return address from the stack top to PC. This ensures return to main program. The Z-80 peripheral chips recognize this instruction and understand that the interrupt service is over. In other respects, RETI is the same as RET instruction.

The following instructions perform the saving of all the general-purpose 8-bit registers and the flags register in the alternate set of registers using only 4 + 4 = 8 clock cycles.

```
EXX
EX AF, AF'
```

This is much faster than using the following instructions that would take up 4 * 11 = 44 clock cycles. In an 8085-based system, the registers had to be saved on the stack only.

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
```

Executing the following instructions would perform restoring of all the general-purpose 8-bit registers and the flags register from the alternate set of registers using only 4 + 4 = 8 clock cycles.

```
EXX
EX AF, AF'
```

This is much faster than using the following instructions that would take up 4 * 10 = 40 clock cycles. Thus the interrupt response is much faster in Z-80 compared with 8085.

```
POP HL
POP DE
POP BC
POP AF
```

Thus the general format of an INT* ISS will be as shown in Fig. 27.13. In case index registers are to be used in the service routine, they also have to be saved and restored using PUSH Iz and POP Iz instructions in the service routine.

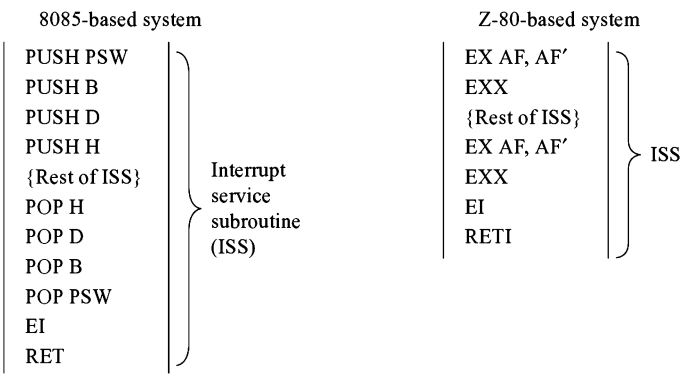


Fig. 27.13
General format for INT* service routine

Interrupt mode 1 (IM 1): This is actually a 2-byte instruction of Z-80 with the opcode ED 56H. After the execution of this instruction the INT* input of Z-80 works in interrupt mode 1. INT* in IM 1 mode of operation is very similar to RST 5.5, RST 6.5, or RST 7.5 which are the vectored interrupts of 8085.

The Z-80 knows the location to branch in this case that is understood to be 0038H. Hence there is no activation of IORQ* and M1* in IM 1 mode.

The Z-80 will automatically disable the interrupt system. The user is not required to explicitly use DI instruction. The Z-80 will save the address of the next instruction (same as PC contents) above the stack top. The Z-80 then branches to the ISS at the specific location 0038H. Hence the action taken by Z-80 when interrupted half way through an instruction because of INT* in IM 1 mode could be summarized as follows. It is assumed that NMI* is not active and interrupt system was enabled using EI instruction.

- Completes the execution of the instruction under progress.
- Disables the interrupt system.
- Saves PC value (address of next instruction) on the stack top.
- Branches to the ISS at 0038H.

In Z-80 kit, there could be an unconditional jump to a system RAM location as part of the monitor program at location 0038H. For example, the instruction 'JP 8FF3H' could be present in the three locations starting from 0038H. Again at location 8FF3H there could be an unconditional jump to a user RAM location as part of the user program. For example, the instruction 'JP 9C00H' could be present in the three locations starting from 8FF3H. In such a case, the INT* service routine in IM 1 mode starts from location 9C00H, as indicated in Fig. 27.14.

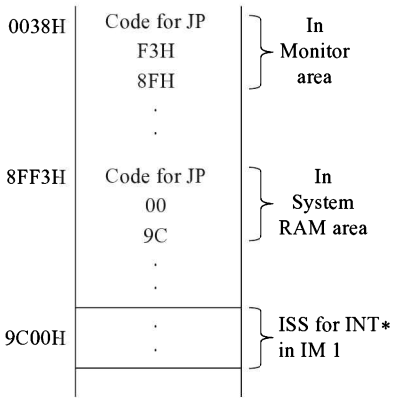


Fig. 27.14
Branching to service routine in IM 1 mode

This mode is functionally the same as IM 0 mode if the peripheral sends RST 38H instruction or CALL 0038H instruction. The advantage of IM 1 mode is that no external hardware is needed to supply the service routine address. It is useful in applications that have a single interrupt source. The disadvantage is that if several interrupt sources are present, there is no direct way in this mode to identify the source of interrupt.

Interrupt mode 2 (IM 2): This is actually a 2-byte instruction of Z-80 with the opcode ED 5EH. After the execution of this instruction the INT* input of Z-80 works in interrupt mode 2. INT* in IM 2 mode of operation has no parallel in 8085. The advantage of IM 2 mode is that as many as 128 interrupting sources could be present in the system. Each interrupting source sends a unique 8-bit address to the Z-80 in response to simultaneous activation of IORQ* and M1*. Based on the 8-bit address received the Z-80 branches to one of the 128 possible service routines. It is a very powerful mode indeed.

The IM 2 mode resembles both IM 0 and IM 1 modes to some extent. The response of Z-80 for activation of INT* in IM 2 mode is explained with an example.

If INT* is activated when Z-80 is in IM 2 mode, then the Z-80 will complete the execution of the instruction in progress. The Z-80 will then respond to the interrupt request by activating IORQ* and M1* signals. The Z-80 expects the interrupting peripheral to respond with an 8-bit address. Hence, the peripheral should send an 8-bit address to the Z-80 using the data lines of Z-80. Let us say 09H = 00001001 is received by the Z-80. The Z-80 always treats the LS bit of this address as 0 immaterial of what is received. Thus, the 8-bit address received is treated as 00001000 = 08H. This is interpreted as the LS byte of a 16-bit address pointer. The MS byte of this 16-bit address pointer is understood to be provided by I register contents. I register stands for 'interrupt vector register' which is loaded using the instruction 'LD I, A'. If the I register content is 05H, then 0508H is taken as a 16-bit address pointer. If I register contains 05H, the Z-80 treats memory locations 0500H to 05FFH to provide an interrupt vector table. In this table, which contains 256 memory locations, 128 ISS addresses, each of 2 bytes, are stored. Assume that in locations 0508H and 0509H the bytes 12H and 34H, respectively are stored as shown in Fig. 27.15. In such a case, the Z-80 branches to the service routine at location 3412H. In this manner, the Z-80 can branch to any one of the 128 interrupt service routines based on the 8-bit address pointer it receives from the peripheral.

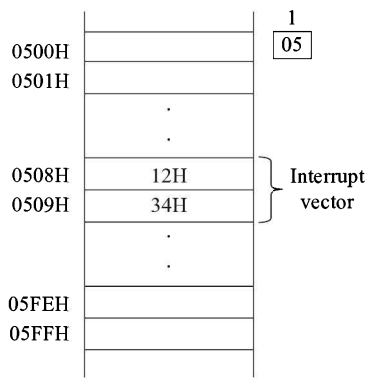


Fig. 27.15
Interrupt vector table
in IM 2 mode

Thus, the action taken by Z-80 when interrupted half way through an instruction because of INT* in IM 2 mode could be summarized as follows. It is assumed that NMI* is not active and the interrupt system is enabled using EI instruction.

- Completes the execution of the instruction under progress.
- Activates IORQ* and M1*.
- Receives an 8-bit address pointer from the peripheral.
- Treats this as LS byte of address pointer after making LS bit as 0.
- Contents of I register is treated as MS byte of address pointer.
- Disables the interrupt system.
- Saves PC value (address of next instruction) on the stack top.
- Branches to the ISS pointed by the address pointer.

27.7.2 NMI* Interrupt

The NMI* is negative edge triggered interrupt input. It has higher priority over INT*. Also, whenever the NMI* pin is activated, the Z-80 always responds by executing the appropriate ISS. In other words,

this interrupt is non maskable. It is a vectored interrupt, very similar to the TRAP interrupt input of 8085. It always branches to the service routine at location 0066H. NMI* interrupt is normally used for high priority events such as power failure, and real time clock. After return from the NMI service routine the interrupt system will be restored to the earlier enabled or disabled status.

The action performed by Z-80 when interrupted half way through an instruction because of NMI* could be summarized as follows.

- Completes the execution of the instruction under progress.
- Disables the interrupt system.
- Saves PC value (address of next instruction) on the stack top.
- Branches to the ISS at location 0066H.

In Z-80 kit, at location 0066H there could be an unconditional jump to a system RAM location as part of the monitor program. For example, the instruction 'JP 8FF0H' could be present in the three locations starting from 0066H. Again at location 8FF0H there could be an unconditional jump to a user RAM location as part of the user program. For example, the instruction 'JP 8C00H' could be present in the three locations starting from 8FF0H. In such a case, the NMI* service routine starts from location 8C00H.

The format of the NMI* service routine will generally be as follows.

- Save all the registers above the stack top.
- Perform the action needed to satisfy the interrupting device.
- Restore all the register contents from the stack top.
- Return to main program by executing RETN instruction.

RETN stands for '**RET**urn from **N**MI service routine'. The RETN instruction does the following.

- Pop from the stack top to PC. This ensures return to main program.
- Get back to old value of interrupt enable/disable status.

Executing the following instructions could perform the saving of all the registers. It is to be noted that the registers are being saved on the stack instead of alternate registers. This is because, the NMI* can get activated when the Z-80 is half way through INT* service routine. In such a case, the Z-80 should return from NMI* service routine to INT* service routine with the contents of main and alternate set of registers unchanged.

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY

EX AF, AF'
EXX
PUSH AF
PUSH BC
PUSH DE
PUSH HL
```

Executing the following instructions would perform restoring of all the registers by popping in reverse order.

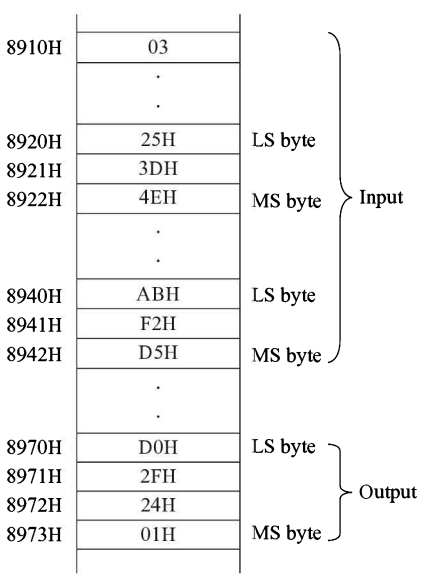
```
POP HL
POP DE
POP BC
POP AF

EX AF, AF'
EXX

POP IY
POP IX

POP HL
POP DE
POP BC
POP AF
```

■ 27.8 PROGRAMMING EXAMPLES

27.8.1 Addition of Multi-Byte Numbers

Write an Z-80 assembly language program to add two multi-byte numbers. The size in bytes of the two multi-byte numbers is provided in location 8910H. The first multi-byte number starts from location 8920H and the second one from location 8940H. The result is to be stored from location 8970H. For the result, one extra location is provided to account for the carry generated.

A typical input and the expected output are indicated in Fig. 27.16.



$$\begin{array}{r} {\scriptstyle 1}\\ 4E\ 3D\ 25H \\ +\ D5\ F2\ ABH \\ \hline 24\ 2F\ D0 \end{array}$$

Fig. 27.16
Addition of two multi-byte numbers

Program

```
        SCF
        CCF

        LD HL, 8910H
        LD IX, 8920H

AGAIN:  LD A, (IX + 0)
        ADC A, (IX + 20H)
        LD (IX + 50H), A

        INC IX
        DEC (HL)
        JP NZ AGAIN

        JP NC EXIT
        LD (IX + 50H), 01
        HALT

EXIT:   LD (IX + 50H), 00
        HALT
```

The reader is urged to compare this program with the one in Sect. 15.2 that used 8085 processor. A shortage of registers was felt while using the 8085 processor. The usage of indexed addressing in Z-80 has resulted in a more compact program, and a number of registers of Z-80 are still available for use. This amply demonstrates the power of indexed addressing mode.

• 27.8.2 Exchange of Blocks

Write an Z-80 assembly language program to exchange the contents of two blocks in memory. The number of elements in the block is provided in location 8910H. The two blocks start at locations 8920H and 8940H, respectively.

A typical input and the expected output are indicated in Fig. 27.17.

	Before	After
8910H	04	04
	.	.
	.	.
8920H	11H	01H
8921H	22H	02H
8922H	33H	03H
8923H	44H	04H
	.	.
	.	.
8940H	01H	11H
8941H	02H	22H
8942H	03H	33H
8943H	04H	44H

Fig. 27.17
Exchange of blocks

Program

```
        LD HL, 8910H
        LD IX, 8920H
AGAIN:  LD A, (IX + 0)
        LD B, (IX + 20H)
        LD (IX + 00H), B
        LD (IX + 20H), A
        INC IX
        DEC (HL)
        JP NZ AGAIN
        HALT
```

The reader is urged to compare this program with the one in Sect. 15.1 that used 8085 processor. The usage of indexed addressing in Z-80 has resulted in a more compact program, once again demonstrating the power of indexed addressing mode.

27.8.3 Block Movement Without Overlap

Write an Z-80 assembly language program to perform block movement. The blocks are assumed to be non overlapping. The block starting at location 8920H is to be moved to that starting at location 8940H. The block size is provided in location 8910H.

A typical input and the expected output are indicated in Fig. 27.18.

	Before	After
8910H	03	03
	.	.
	.	.
8920H	22H	22H
8921H	33H	33H
8922H	44H	44H
	.	.
	.	.
8940H	01H	22H
8941H	02H	33H
8942H	03H	44H

Fig. 27.18
Block movement

Program

```
        LD HL, 8910H
        LD IX, 8920H
AGAIN:  LD A, (IX + 0)
        LD (IX + 20H), A
```

```
INC IX
DEC (HL)
JP NZ AGAIN

HALT
```

The reader is urged to compare this program with the one in Sect. 15.4 that used 8085 processor. Once again, the use of indexed addressing in Z-80 has resulted in a more compact program.

■ 27.9 INSTRUCTION SET SUMMARY

The Z-80 instruction set summary is provided in Table 27.7. The convention used in this table is as follows.

Table 27.7 The Z-80 Instruction Set Summary

Mnemonic	*No. of opcodes*
ADD A/ADC A/SUB/SBC A/AND/OR/XOR/CP r/(HL)/(Iz+a8)/d8	8 * (7 + 1 + 2 + 1) = 88
INC/DEC r/(HL)/Rp/(Iz+a8)	2 * (7 + 1 + 6 + 2) = 32
ADD/ADC/SBC HL, rp	3 * 4 = 12
ADD Iz, zp	2 * 4 = 8
DAA	1
LD Rp, d16	6
LD r1, r2	7 * 7 = 49
LD r | (HL)	7 * 2 = 14
LD r/(HL)/(Iz+a8), d8	7 + 1 + 2 = 10
LD (BC)/(DE)/I/R | A	4 * 2 = 8
LD A | (a16)	2
LD r | (Iz+a8)	7 * 2 * 2 = 28
LD Rp | (a16)	6 * 2 = 12
LD HL | (a16)	1 * 2 = 2
LD SP, HL/IX/IY	3
EX DE, HL; EXX; EX AF, AF′;	3
EX (SP), HL/IX/IY	3
IN A, (a8); OUT (a8), A;	2
IN r, (C); OUT (C), r;	7 + 7 = 14
INI/INIR/INI/INDR/OUTI/OTIR/OUTD/OTDR	8
LDI/LDIR/LDD/LDDR	4
CPI/CPIR/CPD/CPDR	4
BIT/SET/RES*n*, r/(HL)/(Iz + a8)	3 * 8 * (7 + 1 + 2) = 240
RLC/RRC/RL/RR/SLA/SRA/SRL r/(HL)/(Iz+a8)	7 * (7 + 1 + 2) = 70
RLCA/RRCA/RLA/RRA	4
RLD/RRD	2
CALL/JP a16	2
CALL/JP cond, a16	2 * 8 = 16
JP (HL)/(IX)/(IY)	3
DJNZ/JR r8	2
JR C/NC/Z/NZ r8	4
RET cond	8
RET/RETI/RETN	3
RST 0/ 8/ 10H/ 18H/ 20H/ 28H/ 30H/ 38H	8
IM 0/1/2	3
PUSH/POP BC/DE/HL/AF/IX/IY	2 * 6 = 12
CPL/NEG/CCF/SCF/HALT/NOP/EI/DI	8

d16 = 16-bit immediate data;
a16 = 16-bit address;
a8 = 8-bit address;
r8 = 8-bit signed displacement.

r, r1, r2 = A, B, C, D, E, H, or L;
Iz = IX or IY;
rp = BC, DE, HL, or SP;
Rp = BC, DE, HL, SP, IX, or IY;
zp = BC, DE, Iz, SP
that is, if Iz = IX then zp = BC, DE, IX, SP and if Iz = IY then zp = BC, DE, IY, SP;
n = 0, 1, 2, ..., 7;
cond = C/NC/Z/NZ/P/M/PO/PE.

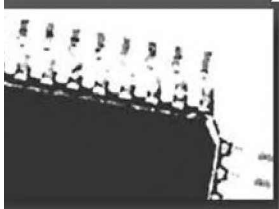
1. Explain the improvements in 8085 over 8080.
2. Provide a brief description of the programmer's view of Z-80.
3. Explain the flags of Z-80.
4. Explain relative addressing in Z-80 and its advantages. List the various conditional relative jump instructions of Z-80.
5. Explain indexed addressing mode of Z-80 with an example.
6. Write the mnemonic in Z-80 for the instructions whose purpose is indicated as follows. Also indicate the instruction size in each case.

   a. Instruction to set bit 5 of D register.
   b. Instruction to reset bit 6 of memory location pointed by HL.
   c. Instruction to test bit 7 for a zero value of memory location 4560H, when IX register is 4500H, using indexed addressing.

7. Write a short note on the opcode size of Z-80 instructions.
8. Explain the difference in the functioning of DAA instruction in 8085 and Z-80.
9. Write a note on the I/O instructions in Z-80.
10. Explain the shift instructions available in Z-80 with examples.
11. With a neat functional pin diagram explain the pins of Z-80.
12. Explain the response by Z-80 when NMI* interrupt is activated.
13. Explain the various modes of operation of INT* in Z-80.
14. Explain the following instructions of Z-80 with examples.

a. EXX	b. EX AF, AF′	c. DJNZ BACK	d. LDIR	e. LDD	f. CPDR
g. INIR	h. IND	i. OTDR	j. OUTI	k. RLD	l. RRD

28



Motorola M6800 Microprocessor

- Pin description of 6800
- Programmer's view of 6800
- Addressing modes of 6800
  - *Immediate addressing*
  - *Implied or inherent addressing*
  - *Direct addressing*
  - *Extended addressing*
  - *Indexed addressing*
  - *Relative addressing*
- Instruction set of 6800
  - *Data transfer group*
  - *Arithmetic group*
  - *Logical group*
  - *Branch group*
  - *Miscellaneous instructions*
- Interrupts of 6800
- Programming examples
  - *Addition of multi-byte numbers*
  - *Exchange of blocks*
  - *Block movement without overlap*
- Questions

In the previous chapters Intel 8085 and Zilog Z-80 microprocessors were dealt with in depth. In this chapter Motorola M6800, another very popular 8-bit microprocessor will be discussed. This will provide the reader with a comparison of the most popular 8-bit microprocessors.

■ 28.1 PIN DESCRIPTION OF 6800

Motorola 6800 is another popular 8-bit microprocessor, with which, the following chips are normally used to provide the required functions in a microcomputer.

—6870 Clock generator;
—6830 ROM or 68708 EPROM;
—6810 RAM;
—6820 Peripheral interface adapter (similar to 8255 PPI in 8085 system);
—6850 Asynchronous communications interface adapter (similar to 8251 USART);
—6828 Priotity interrupt controller (similar to 8259 PIC).

The 6800 is available as a 40-pin DIP and its actual pin diagram is shown in Fig. 28.1. The functional pin diagram is shown in Fig. 28.2.

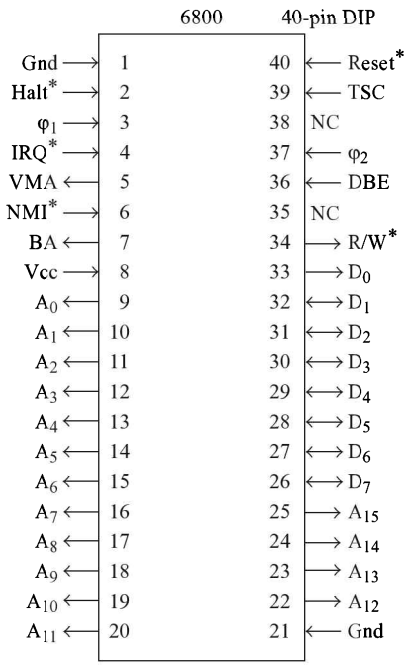


Fig. 28.1
Pin diagram of 6800

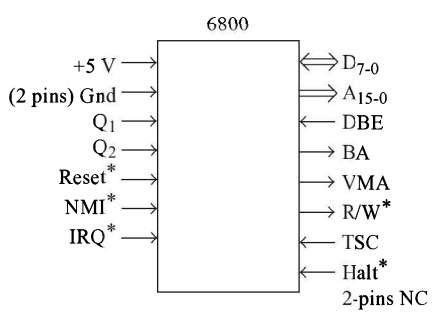


Fig. 28.2
Functional pin diagram of 6800

$A_{15\text{-}0}$: Unidirectional 16-address pins on which address is sent out by 6800. Thus, 6800 is capable of addressing 2^{16} = 64K memory locations.

$D_{7\text{-}0}$: Bi-directional 8-data pins on which data is sent out or received by 6800. Thus, 6800 is an 8-bit microprocessor.

V_{cc}, Gnd: Power supply pins. 6800 uses a dc power supply of +5 V. There are actually two Gnd pins and both of them have to be connected to Gnd.

φ1, φ2: Motorola 6800 receives two-phase clock inputs on these pins. A clock generator chip M6870 is used to provide the two-phase clock. The typical frequency of operation is 1 MHz, while the minimum is 100 kHz. If operated at a frequency lower than 100 kHz, the clock duration would be greater than 10 μs, and some of the dynamic internal registers would lose their data.

R/W*: Active-low output pin. Motorola 6800 sends out a 1 on this pin when reading of data from memory or input port is desired. It sends out a 0 on this pin when a write operation to memory or an output port is desired and uses memory mapped I/O for addressing of I/O ports. As such, there is no pin like IO/M*.

Reset*: Active-low input pin. When brought down to 0 state momentarily, the 6800 is reset.

TSC, BA: TSC is an active-high input pin. TSC stands for tristate control. It is similar in function to hold input of 8085. BA is an active-high output pin. BA stands for bus available. It is similar in function to Hlda output of 8085. These two pins are used for facilitating direct memory access.

NMI*, IRQ*: These are active low interrupt input pins. NMI stands for non-maskable interrupt. IRQ stands for interrupt request that could be masked or disabled. These interrupts are discussed in detail later.

Halt*: Active-low input pin. When this pin is activated the 6800 goes to the halt state. In this state $A_{15\text{-}0}$, $D_{7\text{-}0}$, and R/W* go to their high impedance state and the processor halts. This pin could be used for single stepping through a program.

VMA: Active-high output pin. VMA stands for valid memory address. When this pin is activated by the 6800, it means that the address on $A_{15\text{-}0}$ is a stable valid address. It is activated a little after the address is sent out on $A_{15\text{-}0}$.

DBE: Active-high input pin. DBE stands for data bus enable. When this pin is activated, the data pins $D_{7\text{-}0}$ are enabled. In such a case, the data present on $D_{7\text{-}0}$ enters the 6800 if it is a read operation, or data comes out on $D_{7\text{-}0}$ if it is a write operation. When the pin is in the 0 state, the data pins $D_{7\text{-}0}$ are tristated. It is generally connected to φ2 clock.

NC: There are two NC pins. NC stands for no connection. Thus, the required functions of 6800 are implemented using only 38 pins.

■ 28.2 PROGRAMMER'S VIEW OF 6800

From a programmer's point of view the 6800 is a very simple microprocessor. The programmer's view is indicated in Fig. 28.3a. The simplified internal architecture of 6800 is provided in Fig. 28.3b.

From a programmer's viewpoint the 6800 has:

- Two 8-bit accumulators—A and B;
- Three 16-bit address registers—IX, SP and PC;
- Eight-bit flags register—CCR.

IX register is used as index register. It is shown as X in an instruction mnemonic.
SP is the stack pointer. It is shown as S in an instruction mnemonic.
PC is the program counter.

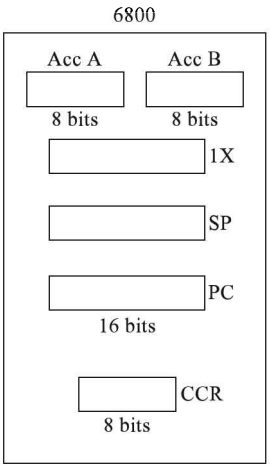


Fig. 28.3a
Programmer's
view of 6800

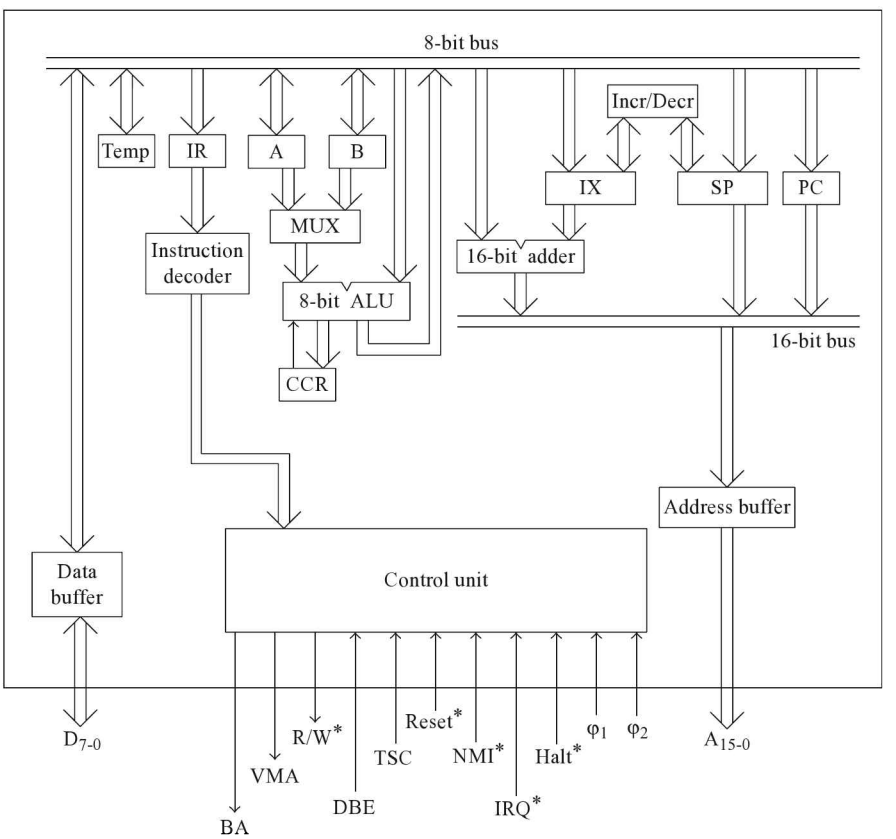


Fig. 28.3b
Simplified
architecture
of 6800

Both A and B registers have almost the same prominence in the instruction set. This was not the case with 8085 where A register was more important than other 8-bit registers. In 6800 there are only a few operations that can be done with A, but not by B. The following list shows the exceptions.

ABA (add B and A, store result in A) is implemented, but not AAB;
SBA (subtract B from A, store result in A) is implemented, but not SAB;

CBA (compare B with A, store result in a temporary register) is implemented, but not CAB;
TAP (transfer from A to CCR) is implemented, but not TBP;
TPA (transfer from CCR to A) is implemented, but not TPB.

Flags register: The flags register in 6800 is termed CCR that stands for condition code register. The flags register contents are indicated in the following. It contains only six flags in the LS position, of which five are status flags and the remaining one is a control flag. The other two bits in the MS bit positions of the CCR are always in the 1 state.

1	1	H	IM	N	Z	V	Cy

The status flags are affected based on the result of execution of an instruction. The five status flags are:

H flag: It is half carry flag and is the same as the auxiliary carry of 8085.
N flag: It is negative flag and is the same as the sign flag of 8085.
Z flag: It is zero flag.
Cy flag: It is carry flag.
V flag: It is overflow flag and is described in Z-80.

Note that the P flag of 8085 is omitted here, as it is not a useful flag.

The control flag in CCR is the IM flag. IM stands for interrupt mask. If this flag is set to 1, then IRQ* interrupt will be masked or disabled. It implies that, the 6800 will not be interrupted even if the IRQ* pin is activated by a peripheral. If reset to 0, the IRQ* interrupt will be unmasked or enabled. That is, if the IRQ* pin is activated by a peripheral, the 6800 will be interrupted, provided of course, the higher priority NMI* is not active at the same time.

The IM bit is reset to 0 by the execution of the instruction CLI that stands for CLear Interrupt mask bit. The CLI instruction is functionally the same as the EI instruction of 8085. The IM bit is set to 1 by the execution of the instruction SEI that stands for SEt Interrupt mask bit. The SEI instruction is functionally the same as the DI instruction of 8085. The CCR register is indicated as P in an instruction mnemonic for brevity. For example, the mnemonic TAP stands for transfer from A register to P (CCR) register.

■ 28.3 ADDRESSING MODES OF 6800

There are six addressing modes in 6800. They are:

Immediate addressing;
Implied or inherent addressing;
Direct addressing or page 0 addressing;
Extended addressing or absolute addressing;
Indexed addressing;
Relative addressing.

The following conventions are used in the assembly language programming of 6800.

A value 3FH is indicated as $3F. Thus, $ stands for an hexadecimal number. With $ not indicated, the number is taken to be a decimal number.

Immediate data in an instruction follows '#' symbol. A value that is not preceded by '#' symbol is taken to be a memory address.

If there are two operands for an instruction, the source operand appears first, and the destination operand appears later in the instruction. For example, the mnemonic SBA stands for 'subtract B from A, and store the result in A'. This is exactly the opposite of the situation in 8085 or Z-80.

The 16-bit data or memory address in a 3-byte instruction is stored in memory with MS byte at a lower memory address, and LS byte at a higher address. It is not stored in byte reversal form that was prevalent in 8085 and Z-80. For example, the instruction STA A $1234 is stored in memory as B7 12 34, where B7 is the opcode for STA A.

28.3.1 Immediate Addressing

In this addressing, the data to be used in the execution of the instruction is provided in the instruction itself, immediately after the opcode of the instruction. Some examples follow.

```
LDA B #$3F;   Load accumulator B with the immediate data 3FH.
CPX #$1234;   Compare index register contents with the immediate data 1234H.
LDS #$3456;   Load SP with the immediate data 3456H.
```

28.3.2 Implied or Inherent Addressing

If the operands used in the instruction are within the 6800, the addressing mode is called implied or inherent addressing mode in 6800 terminology. Thus in this addressing mode, the operands will not be in memory. Some examples follow.

```
ABA;    Add B and A and store the result in A.
ASL B;  Arithmetic shift left B register.
CBA;    Compare B contents with A contents. Only flags are affected.
CLC;    Clear Cy flag to 0.
```

28.3.3 Direct Addressing

In this mode the operand will be in memory. This mode is sometimes called page 0 addressing because the operand in memory is restricted to be in the address range of 0000H to 00FFH that happens to be page 0 of memory. Memory in 6800 system can be thought of as made up of pages of size 100H = 256 bytes. Some examples follow.

```
ADD B, $25;   Add contents of location 0025H to B register. It is a 2-byte instruction.
STA B, $F0;   Store B contents in location 00F0H. It is a 2-byte instruction.
LDX $45;      Load IX from contents of locations 45H and 46H. It is a 2-byte instruction.
```

Direct addressing is a very powerful addressing mode provided by 6800. This was not present in 8085 or Z-80. Instructions that use direct addressing will be only 2-byte long compared to the 3-byte length of instructions that use extended addressing. As such, direct addressing instruction occupies less space in memory. And more importantly, it takes less time to fetch and execute an instruction that uses direct addressing.

In a 6800 system it is advantageous to have the frequently accessed data stored in the memory range 0000H–00FFH. The data can hence be accessed very fast using direct addressing. To facilitate this, RAM generally starts from location 0000H and grows towards higher addresses in the memory space of 6800. ROM or EPROM is provided the last part of the memory space, and ends with the address FFFFH. This is shown in Fig. 28.4.

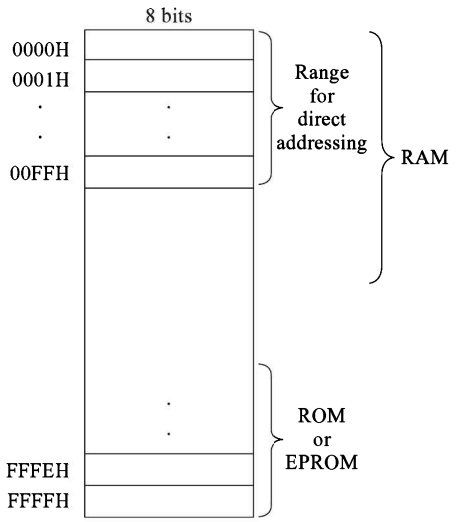


Fig. 28.4 Memory space in 6800

28.3.4 Extended Addressing

It is the same as the absolute addressing of 8085. However, it is much more powerful here. In this mode the operand will be in memory. The address of the operand is provided after the opcode in the instruction using 16 bits. Because of the 16-bit address, the operand could be anywhere in the memory space in the range 0000H to FFFFH. Some examples follow.

SBC B $2345;	Subtract from B with carry the contents of location 2345H. Store result in B.
CLR $ABCD;	Clear the contents of memory location ABCDH.
TST $AABB;	Test contents of location AABBH for 00H value.
NEG $6677;	Negate the contents of location 6677H. Contents changed to FEH (= −02) if the original contents of 6677H was 02H.
COM $7788;	Complement (1's complement) the contents of location 7788H. Contents changed to FDH if the original contents of 7788H was 02H.

It may be noted that none of these instructions were present in the absolute addressing mode of 8085. This demonstrates the power of extended addressing in 6800.

If the user writes in assembly language an instruction like COM $0056, the 6800 assembler automatically generates a 2-byte instruction code treating the instruction as COM $56 in direct addressing mode. As such, the practical addressing range for extended addressing could be indicated as 0100H to FFFFH.

28.3.5 Indexed Addressing

In this mode the operand will be in memory. A part of the address is provided in the IX register and the other part of size 8 bits is provided directly in the instruction. The sum of these two parts provides

the complete memory address for the operand. This mode is already described in Z-80. However, in 6800 there is a single IX register compared to the two present in Z-80. Some examples follow. In these instructions, the IX content is assumed to be 1234H.

ADC B $10, X;	Add with carry to B register the contents of memory location 1244H.
STS $30, X;	Store SP contents in memory locations 1264H and 1265H.
BIT B $25, X;	AND contents of B register with contents of memory location 1259H. Contents of B remain unchanged. Only flags are affected.
CPX $20, X;	Compare IX contents with contents of memory locations 1254H and 1255H.

• 28.3.6 Relative Addressing

The relative addressing mode is used in the branch group of instructions. The branch group of instructions that use relative addressing will be only 2-byte long and will be executed faster compared to those that use extended addressing. In relative addressing, a signed 8-bit number will follow the opcode byte. This number indicates whether the branch is to a forward location or to a backward location. The range for branching is 127 locations forward or 128 locations backward from the next instruction. This mode is already described in Z-80. Some examples follow.

BRA $F0;	Branch unconditionally 10H locations backward (F0H = −10H) from the next instruction after BRA $F0.
BEQ $F0;	Branch if equal to zero (Z flag = 1) 10H locations backward (F0H = −10H) from the next instruction after BEQ $F0.
BVS $20;	Branch if V flag is set to 20H locations forward from the next instruction after BVS $20.
BSR $20;	Branch to subroutine that is 20H locations forward from the next instruction after BSR $20.

■ 28.4 INSTRUCTION SET OF 6800

Some statistics about the instruction set of 6800 are shown in the following.

- Instruction types: 72;
- Number of opcodes: 197;
- Instruction length: 1, 2, or 3 bytes.

Instruction length (bytes)	*Number of opcodes*
1	51
2	103
3	43
	Total 197 opcodes

Although 6800 has only 197 opcodes compared to the 246 of 8085, it is more powerful than 8085. Its branch group of instructions are more powerful than even Z-80 with about 700 opcodes. The instruction set of 6800 is easy to master considering its simplicity. The power of 6800 instruction set will be demonstrated by some programming examples later. The instructions of 6800 could be classified under the following groups.

Instruction class	*No. of opcodes*	*No. of instruction types*
Data transfer group	38	14
Arithmetic group	55	15
Logical group	73	14
Branch group	23	21
Miscellaneous	8	8
	Total 197 opcodes	Total 72 types

The following convention is used in the description of the instruction set of 6800.

d8 = 8-bit data;
a8 = 8-bit address;
a16 = 16-bit address;
IX+a8 = Indexed addressing using IX and 8-bit displacement;
r8 = 8-bit relative (signed) displacement.

LDA/STA = Load/store accumulator;
LDS/STS = Load/store stack pointer;
LDX/STX = Load/store IX register;
TAP = Transfer from A to P;
PSH/PUL = Push/pull (pop).

28.4.1 Data Transfer Group

There are 38 opcodes in this group under 14 instruction types.

Mnemonic	*No. of opcodes*	*No. of instruction types*
LDA A/B, d8/a16/a8/IX+a8	2 * 4 = 8	1
STA A/B, a8/a16/IX+a8	2 * 3 = 6	1
LDS/LDX d16/a16/a8/IX+a8	2 * 4 = 8	2
STS/STX a8/a16/IX+a8	2 * 3 = 6	2
PSH/PUL A/B	2 * 2 = 4	2
TAB/TBA/TSX/TXS/TAP/TPA	6	6

Note: TBP/TPB are not implemented.

28.4.2 Arithmetic Group

There are 55 opcodes in this group under 15 instruction types.

Mnemonic	*No. of opcodes*	*No. of instruction types*
SBC/SUB/ADC/ADD A/B d8/a8/a16/IX+a8	4 * 2 * 4 = 32	4
ABA/SBA	2	2
INS/INX/DES/DEX	4	4
NEG/CLR/INC/DEC A/B/a16/IX+a8	4 * 4 = 16	4
DAA	1	1

Note: 'NEG/CLR/INC/DEC a8' are not implemented.

28.4.3 Logical Group

There are 73 opcodes in this group under 14 instruction types.

Mnemonic	*No. of opcodes*	*No. of instruction types*
BIT/CMP/EOR/ORA/AND A/B, d8/a8/a16/IX+a8	5 * 2 * 4 = 40	5
ROR/ROL/LSR/ASR/ASL/COM/TST A/B/a16/IX+a8	7 * 4 = 28	7
CPX d16/a8/a16/IX+a8	4	1
CBA	1	1

Notes:
ROR (rotate right) and ROL (rotate left) operations are on 8 bits only. There are no 9-bit rotations including carry similar to the ones in 8085.
ASR and ASL stand for arithmetic shift right and left respectively.
LSR stands for logical shift right. There is no LSL as it would be the same as ASL.
Also, 'ROR/ROL/LSR/ASR/ASL/COM/TST a8' are not implemented.
TST stands for Test. It subtracts 00H from the operand specified in the instruction. Flags are affected.
BIT stands for Bit Test. It performs logical AND of A/B with the operand. Only flags affected Neither A/B nor operand is affected.

• 28.4.4 Branch Group

There are 23 opcodes in this group under 21 instruction types. The conditional branch instructions of even Z-80 performed branch based on the value of a single flag, except V flag. For example, 'JR NC, r8' of Z-80 performed a branch only if Cy flag were 0. In 6800, conditional branch instructions that test V flag are also provided. They are very useful when working with signed numbers and are shown as follows:

BVS—branch if V flag is set
BVC—branch if V flag is cleared.

In addition, there are conditional branch instructions that test more than one flag also in 6800. Such instructions are useful in the comparison of two unsigned or two signed numbers.

In 6800 terminology, 'higher' and 'lower' are used when comparing unsigned numbers. 'Greater than' and 'less than' are used when comparing signed numbers. Consider the numbers 85H and 45H as an example. 85H is a negative number (as the MS bit is 1) and 45H is a positive number (as the MS bit is 0) when interpreted as signed numbers. Thus, all the following statements are perfectly true in 6800 terminology.

85H is higher than 45H,
85H is less than 45H,

45H is lower than 85H,
45H is greater than 85H.

BHI (branch if higher) and BLS (branch if lower or same) are used with unsigned numbers. BHI and BLS are generally used after a compare operation like 'CMP B, #$25'. These instructions perform a branch based on the value of Cy flag and Z flag.

BGT (branch if greater than), BGE (branch if greater or equal), BLT (branch if less than) and BLE (branch if less than or equal) are used with signed numbers. They are generally used after a compare operation like 'CMP B, #$25'. These instructions perform a branch based on the value of sign flag, V flag, and Z flag.

The other instructions under the branch group are:

BRA—unconditional relative branch
JMP—unconditional branch using extended or indexed addressing
BSR—unconditional relative branch to subroutine
JSR—unconditional branch to subroutine using extended or indexed addressing
RTS—return from subroutine
RTI (return from ISS) and SWI (software interrupt) will be described later.

Note that there are no conditional branches to a subroutine and conditional returns from a subroutine. They are not implemented, as it is possible to manage without such instructions. If it is required to branch to subroutine at symbolic memory location SUBR only if Cy flag is 0, it is managed as follows.

```
BCS NEXT
BSR SUBR
NEXT: next instruction
```

If it is required to return from subroutine only if V flag is 0, it is managed as follows.

```
BVS SKIP
RTS
SKIP: next instruction
```

Mnemonic	*No. of opcodes*	*No. of instruction types*
BCC/BCS/BEQ/BNE/BMI/BPL/BVC/BVS r8	8	8
BHI/BLS r8	2	2
BGT/BGE/BLT/BLE r8	4	4
BRA/BSR r8	2	2
JMP/JSR a16/IX+a8	2 * 2 = 4	2
RTS/RTI	2	2
SWI	1	1

28.4.5 Miscellaneous Instructions

There are eight opcodes in this group under eight instruction types.

Mnemonic	*No. of opcodes*	*No. of instruction types*
WAI	1	1
NOP	1	1
CLC/SEC/CLI/SEI/CLV/SEV	6	6

WAI (wait for interrupt) is functionally the same as the HLT instruction of 8085. There are instructions to clear or set the Cy, IM, and V flags. CLC stands for clear carry, SEC stands for set carry and so on.

6800 instruction set summary

Mnemonic	*No. of opcodes*
BIT/CMP/EOR/ORA/AND/SBC/SUB/ADC/ADD/LDA A/B, d8/a8/a16/IX+a8	10 * 2 * 4 = 80
CPX/LDS/LDX d16/a8/a16/IX+a8	3 * 4 = 12
STA A/B a8/a16/IX+a8	2 * 3 = 6
STS/STX a8/a16/IX+a8	2 * 3 = 6
ROL/ROR/LSR/ASR/ASL/COM/TST/NEG/CLR/INC/DEC A/B/a16/Ix+a8	11 * 4 = 44
PSH/PUL A/B	2 * 2 = 4
TAB/TBA/TSX/TXS/TAP/TPA	6
INS/INX/DES/DEX	4
CBA/ABA/SBA	3
CLC/SEC/CLI/SEI/CLV/SEV	6
BCC/BCS/BEQ/BNE/BMI/BPL/BVC/BVS r8	8
BHI/BLS/BGT/BGE/BLT/BLE r8	6
BRA/BSR r8	2
JMP/JSR a16/IX+a8	2 * 2 = 4
RTS/RTI	2
SWI/WAI/NOP/DAA	4

Note: 'ROL/ROR/LSR/ASR/ASL/COM/TST/NEG/CLR/INC/DEC a8' and CAB/AAB/SAB/TBP/TPB instructions are not implemented.

Speed of instruction execution: The typical clock frequency used in 6800 is 1 MHz. The one used in 8085 is 3 MHz. This gives a wrong impression that 8085 is three times faster than 6800. A simple example is provided to prove it to be a wrong impression.

Consider the execution of the 6800 instruction 'LDA A, #$F3'. Its equivalent in 8085 is 'MVI A, F3H'. In 8085 this instruction takes a total of seven clock cycles for the opcode fetch, decode and execute portion. It works out to 2.33 μs if the 8085 is operating at 3 MHz. In 6800 this instruction takes a total of two clock cycles only—one for fetching and decoding, and the other for execution. It works out to 2 μs if the 6800 is operating at 1MHz. Thus 6800 working at 1 MHz is slightly faster than 8085 processor working at 3 MHz.

Processor	*Instruction*	*Clock speed (MHz)*	*No. of T states*	*Instruction time (μs)*
6800	LDA A, #$F3	1	2	2
8085	MVI A, F3H	2	7	2.33

■ 28.5 INTERRUPTS OF 6800

There are two hardware interrupt pins in 6800 which are NMI* and IRQ*. These are active low interrupt input pins of which NMI* is non maskable. Thus the 6800 gets interrupted whenever the NMI* pin is activated. IRQ* is maskable. It is masked or disabled if the IM flag of CCR register is set to 1

using the SEI instruction. It is also automatically masked whenever the 6800 enters an ISS. In such a case, the 6800 is not interrupted even if the IRQ* pin is activated.

IRQ* is unmasked or enabled if the IM flag of CCR register is reset to 0 using the CLI instruction. In such a case, whenever the IRQ* pin is activated, the 6800 is interrupted, provided NMI* is not active at the same time. This is because, NMI* has a higher priority over IRQ*. When NMI* pin is activated half way through an instruction, the action performed by 6800 is as follows:

The instruction under execution is completed.
PC value saved on the stack.
All the other registers saved on the stack in a particular order as shown in figure 28.5.
IM flag is set to 1.
Jumps to the ISS whose address is provided in locations FFFCH and FFFDH.

If the contents of locations FFFCH and FFFDH are 56H and 78H as shown in Fig. 28.5, then the NMI* ISS starts from 5678H and ends with RTI instruction. The execution of this instruction results in the following.

IM flag is restored to the value that was present prior to interruption.
Register values restored from the stack.
Return to main program by restoring PC value.

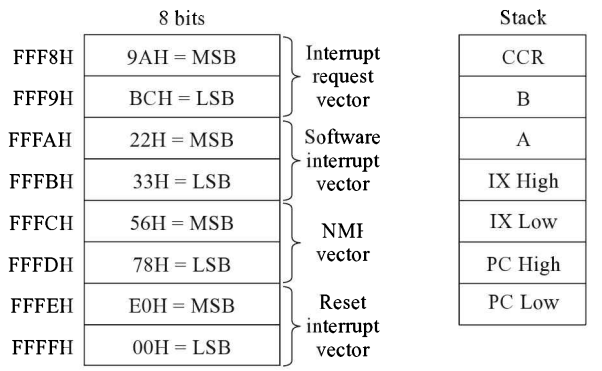


Fig. 28.5
Interrupt vectors of 6800 and saving of registers on the stack

As there is no explicit need for saving and restoring registers in the ISS, the interrupt response is much faster than in 8085. In the ISS of 8085, the user had to explicitly save and restore the registers in the ISS, thereby slowing down the interrupt response.

When IRQ* pin is activated half way through an instruction, the action performed by 6800 is very much similar to the activation of NMI*. In this case the 6800 jumps to the ISS whose address is provided in locations FFF8H and FFF9H. If the contents of locations FFF8H and FFF9H are 9AH and BCH as shown in Fig. 28.5, then the IRQ* ISS starts from 9ABCH and the ISS ends with RTI instruction.

In addition to the two hardware interrupts, there is also an SWI (software interrupt) in 6800. This instruction has the mnemonic SWI. When this instruction is executed by the 6800 in a program, the action taken is as follows.

PC value saved on the stack.
All the other registers saved on the stack in a particular order as shown in figure 28.5.
IM flag is set to 1.
Jumps to the ISS whose address is provided in locations FFFAH and FFFBH.

If the contents of locations FFFAH and FFFBH are 22H and 33H as shown in Fig. 28.5, then the ISS for SWI instruction starts from 2233H and it ends with RTI instruction.

The activation of Reset* pin also interrupts the working of 6800. The 6800 always responds to activation of Reset* signal in the following way.

The instruction under execution is immediately suspended.
IM flag is set to 1.
Jumps to the address that is provided in locations FFFEH and FFFFH.

If the contents of locations FFFEH and FFFFH are E0H and 00H as shown in Fig. 28.5, then the jump takes place to E000H. At this location the system initialization program begins.

■ 28.6 PROGRAMMING EXAMPLES

28.6.1 Addition of Multi-Byte Numbers

Write a 6800 assembly language program to add two multi-byte numbers. The size in bytes of the two multi-byte numbers is provided in location 8910H. The first multi-byte number starts from location 8920H with the LS byte and the second one from location 8940H with the LS byte. The result is to be stored from location 8970H starting with the LS byte. For the result, one extra location is provided to account for the carry generated.

A typical input and the expected output are indicated in Fig. 28.6.

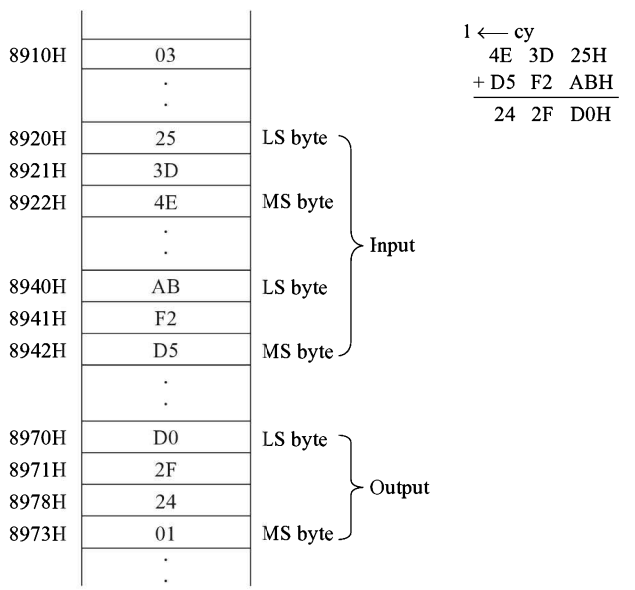


Fig. 28.6
Addition of two multi-byte numbers

Program

```
        CLC
        LDX #$8920

AGAIN:  LDA A $0, X
        ADC A $20, X
        STA A $50, X

        INX
        DEC $8910
        BNE AGAIN

        CLR $50, X
        BCC EXIT
        INC $50, X
EXIT:   WAI
```

The reader is urged to compare this program with the one in Section 14.2 that used 8085 processor. A shortage of registers was felt while using 8085 processor. The number of registers in 6800 is much less than in 8085. In spite of this, in the above program, register B of 6800 is still available for use. This is because, the usage of indexed addressing in 6800 has resulted in a more compact program, which amply demonstrates the power of indexed addressing mode.

28.6.2 Exchange of Blocks

Write a 6800 assembly language program to exchange contents of two blocks in memory. The number of elements in the block is provided in location 8910H. The two blocks start at locations 8920H and 8940H, respectively.

Typical memory contents before and after program execution are indicated in Fig. 28.7.

	Before	After
8910H	04	04
	.	.
	.	.
8920H	11H	01H
	22H	02H
	33H	03H
	44H	04H
	.	.
	.	.
8940H	01H	11H
	02H	22H
	03H	33H
	04H	44H

Fig. 28.7
Exchange of blocks

Program

```
        LDX #$8920

REPEAT: LDA A 0, X
        LDA B $20, X

        STA A $20, X
        STA B 0, X

        INX
        DEC $8910
        BNE REPEAT

        WAI
```

The reader is urged to compare this program with the one in Sect. 14.1 that used 8085 processor. The usage of indexed addressing in 6800 has resulted in a more compact program, which once again demonstrates the power of indexed addressing mode.

28.6.3 Block Movement Without Overlap

Write a 6800 assembly language program to perform block movement. The blocks are assumed to be non overlapping. The block starting at location 8920H is to be moved to the block starting at location 8940H. The block size is provided in location 8910H.

Typical memory contents before and after program execution are indicated in Fig. 28.8.

	Before	After
8910H	03	03
	.	.
	.	.
8920H	22H	22H
	33H	33H
	44H	44H
	.	.
	.	.
8940H	01H	22H
	02H	33H
	03H	44H

Fig. 28.8
Block movement

Program

```
        LDX #$8920

AGAIN:  LDA A 0, X
        STA A $20, X

        INX
        DEC $8910
        BNE AGAIN

        WAI
```

The reader is urged to compare this program with the one in Sect. 14.4 that used 8085 processor. Once again, the use of indexed addressing in 6800 has resulted in a more compact program.

1. With a neat functional pin diagram, explain the pins of 6800.
2. With a neat diagram describe the programmer's view of 6800.
3. Explain the flags available in 6800.
4. Write a note on the addressing modes of 6800 with an example for each addressing mode.
5. List the various instruction types available in the data transfer group of instructions.
6. List the various instruction types available in the arithmetic group of instructions.
7. List the various instruction types available in the logical group of instructions.
8. Write a note on the branch group of instructions.
9. Comment on the speed of execution of instructions in 6800 and 8085.
10. Write a note on the interrupts of 6800.
11. Describe the internal architecture of 6800 with a neat diagram.

29

8051 Microcontroller

- Main features of Intel 8051
- Functional blocks of Intel 8051
- Program memory structure
- Data memory structure
  - *Internal RAM organization*
  - *Internal data memory organization*
  - *SFR area*
- Programmer's view of 8051
- Addressing modes of 8051
  - *Immediate addressing*
  - *Register addressing*
  - *Direct addressing*
  - *Register indirect addressing*
  - *Indexed addressing*
  - *Implied addressing*
- Instruction set of 8051
  - *Data transfer group*
  - *Arithmetic group*
  - *Logical group*
  - *Bit-processing group*
  - *Program branch group*
- Programming examples
  - *Shift a multi-byte BCD number to the right*
  - *Binary to BCD conversion*
  - *BCD to binary conversion*
  - *BCD to binary conversion (alternative)*
  - *Hex to ASCII conversion*
  - *Bit manipulation program*
  - *Conversion of four-digit hex to ASCII*
- Questions

This chapter is devoted to the study of 8051, a popular microcontroller chip of the Intel family. A microprocessor like 8085 will have ALU, control unit, and a few registers. But a microcontroller will typically have RAM, I/O ports, and timers in addition to what is generally found in a microprocessor. Many versions of microcontrollers also have on-chip ROM or EPROM, and UART. Of course, some even have ADC on the chip. Thus a microcontroller generally has all the blocks that make up a microcomputer. Building a complete microcomputer on a single chip reduces the size and cost of the application.

Microcontrollers are used in almost every electrical and electronic gadget, like televisions, VCRs, washing machines, telephones, microwave ovens, compact disk players, computer keyboards, and printers. A microcontroller is viewed as a single-chip solution to most of the small- and medium-type applications. Microcontrollers are obviously more complex than microprocessors in their architecture. Thus they were developed a few years after microprocessors were developed. The first generation of microcontrollers from Intel was 8048. Intel 8051 can be treated as a second-generation microcontroller with improved features. It belongs to MCS-51 family of Intel microcontrollers.

■ 29.1 MAIN FEATURES OF INTEL 8051

The main features of Intel 8051 are listed as follows and details about these features are provided in the rest of this chapter and the next.

- 8-bit CPU optimized for control applications;
- Bit processing capability;
- Separate program memory and data memory spaces (Harvard architecture);
- 4 kilobytes of on-chip EPROM for program memory (range 0000–0FFFH);
- 128 bytes of on-chip data RAM (range 00–7FH);
- 64 kilobytes program memory address space. This includes 4K bytes on-chip program memory;
- 64 kilobytes data memory address space. This is excluding 128 bytes on-chip data RAM;
- 32 bi-directional and individually addressable I/O lines;
- Two numbers of 16-bit timer/counters T0 and T1;
- Full duplex UART;
- On-chip clock oscillator;
- Interrupts from six sources, two external and four internal;
- Two-level interrupt priority structure;
- Security features for EPROM parts against software piracy;
- 255 instructional opcodes, using 111 instruction types;
- 64 instruction types executed in single machine cycle of one microsecond, when crystal frequency is 12 MHz.

Intel 8051 employs Harvard type of architecture. In this type, the program memory and data memory are separated. With this feature, overlapping of program and data memory space is avoided, providing improved security for program and data.

Intel 8751 is another member of the MCS-51 family. It has 4K bytes of on-chip EPROM instead of ROM. Yet another member, Intel 8031 does not have any program memory on-chip. Thus, 8751 is an EPROM version of 8051, and 8031 is a ROM less version of 8051. Apart from the difference mentioned here, these chips have the same instruction set and internal architecture.

■ 29.2 FUNCTIONAL BLOCKS OF INTEL 8051

The 8051 is available in various versions as 8051H, 8051BH, and 80C51. In this book they are referred to in general as 8051. The 8051, 8031, and 8751 are available in 40-pin DIP package as well as 44-pin LCC (leadless chip carrier package with pins on all four sides of the square chip). In the LCC package four pins would be unused pins. The functional pin diagram of 8051 is provided in Fig. 29.1, and the actual pin diagram in Fig. 29.2.

The 8051 works on a power supply of +5-V dc. It has an on-chip clock circuit. All that is needed to complete the clock circuit is to connect a quartz crystal between the pins XTAL1 and XTAL2, and two capacitors as shown in Fig. 29.1. The maximum crystal frequency is typically 12 MHz. There are versions of 8051 that allow maximum crystal frequency of 20 MHz. There is a flip-flop inside 8051 that divides the clock frequency by 2, to provide the internal clock. Thus with a 12-MHz crystal, the internal clock frequency is 6 MHz. An internal clock period constitutes one state. A machine cycle of 8051 constitutes six such states, S1 to S6. Thus if the crystal frequency is 12 MHz, the duration of a machine cycle is 1 μs. Most of the instructions in the instruction set of 8051 are executed in just one machine cycle.

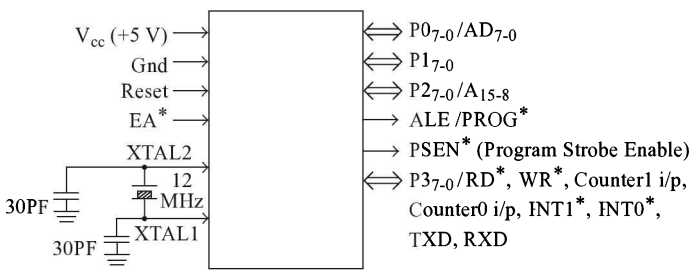


Fig. 29.1
Functional pin diagram of 8051

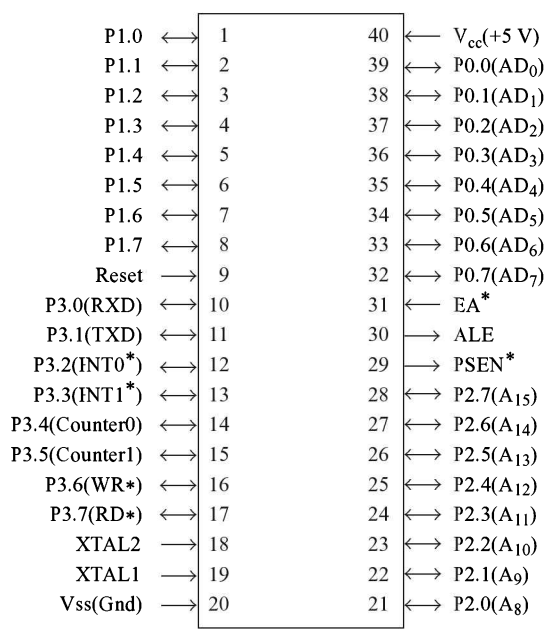


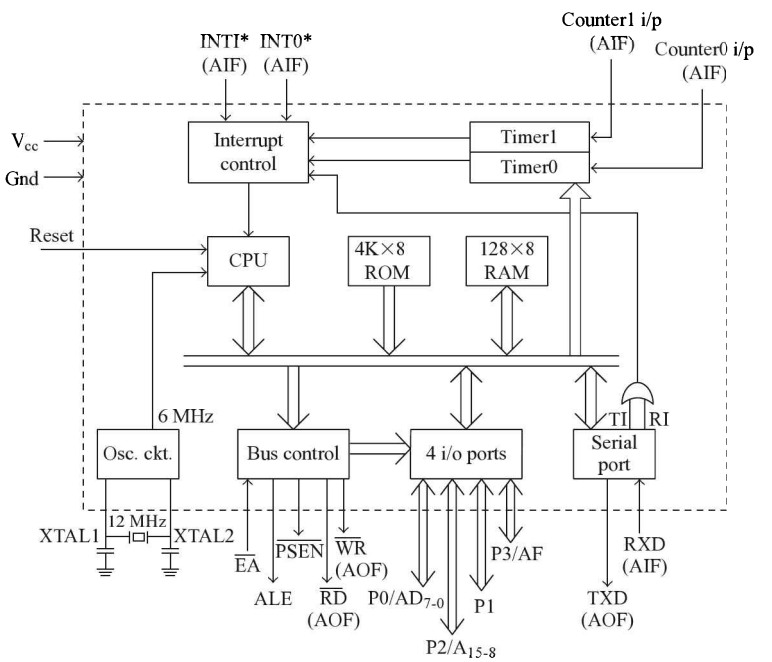
Fig. 29.2
Pin diagram of 8051

It is also possible to drive the 8051 from an external clock source. In such a case, the clock output of the external source is connected to XTAL2 input of 8051 and XTAL1 to ground. Reset is an active high input. Logic 1 on this input for atleast two machine cycles, when the clock is running, resets the 8051.

The 8051 has four 8-bit I/O ports. The ports are called Port 0, Port 1, Port 2, and Port 3. These four ports account for 32 pins of 8051. When the internal data and/or program memory is not adequate for an application, external data and/or program memory will have to be used. In such a case, Port 2 (P2) pins provide the MS byte of address A_{15-0} to external memory, and P0 pins provide the LS byte of address. P0 pins are also used for data/code transfer between 8051 and external data/program memory. Thus P0 pins are used as multiplexed address data pins AD_{7-0}. The ALE output from 8051 indicates whether address or data is present on AD_{7-0} pins. ALE stands for 'address latch enable'. If logic 1 is sent out on ALE, it implies that 8051 is sending out LS byte of address on P0 pins. An external latch is used to latch this LS byte of address.

Two pins of P3 will be used for sending out read (RD*) and write (WR*) signals to external data memory. A signal with a '*' after the signal name indicates that it is an active low signal. Thus RD* and WR* are active low signals. Also, if it is desired to use the internally provided timers, UART, and interrupt features, then the remaining pins of P3 will be used up for this purpose. In such a case, C0 and C1 pins are used as two counter inputs (Intel refers to them as T0 and T1), RXD and TXD pins are used by the UART for receiving and sending serial data, INT0* and INT1* are used as two external interrupt pins. Thus, only P1 is available for I/O operations when external memory is being used.

It is possible to use individually every port pin as input or output. A bit value written to a port bit comes out on the port pin. But if a port pin is to be used as an input pin, the corresponding port bit must be written with a 1 first. Only then must the port pin be read. The reason for this will become clear in the next chapter, where a detailed description of port structures is provided. In fact, after reset of 8051 all the port bits are written with 1s. Figure 29.3 provides the simplified block diagram of 8051.



AF ⟶ Alternate function
AIF ⟶ Alternate input function
AOF ⟶ Alternate output function

TI ⟶ Transmitter interrupt
RI ⟶ Receiver interrupt

Fig. 29.3
Simplified block diagram of 8051

■ 29.3 PROGRAM MEMORY STRUCTURE

The logic level on EA* pin indicates whether the 8051 makes use of the internal program memory or not. EA* stands for 'external access'. If this pin is connected to logic 0, the internal on-chip ROM of 4K byte capacity will not be used. The 8051 in this case accesses only external ROM in the range 0000H–FFFFH. It is to be noted that in 8031, which is ROM less version of 8051, the EA* pin must be connected to ground.

RD* and WR* signals output by 8051 on P3 pins are used for accessing the external data memory only. So in order to facilitate reading from the external ROM, 8051 provides PSEN* output. It is an active low-output pin, which stands for 'program strobe enable'. When PSEN* = 0, the 8051 reads from the external ROM. The 16-bit address of the external ROM location is output on P0 (LS byte) and P2 (MS byte). The program code is received on P0 pins.

If EA* is tied to logic 1, the internal on-chip ROM is used for address range 0000H–0FFFH. External ROM is accessed for addresses in the range 1000H–FFFFH. Thus it can be concluded that the total amount of program memory can be a maximum of 64K bytes including the internal program memory. The external program memory is accessed in the following cases.

(a) When EA* is tied to logic 0.
(b) When program memory in the address range 1000H–FFFFH is accessed, immaterial of logic level on EA* input.

Figures 29.4 and 29.5 provide program memory structure and interfacing of external program memory, respectively.

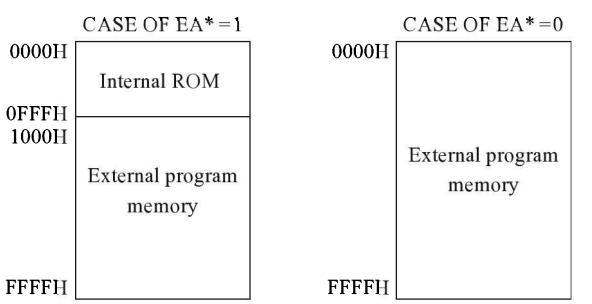


Fig. 29.4
Program memory structure

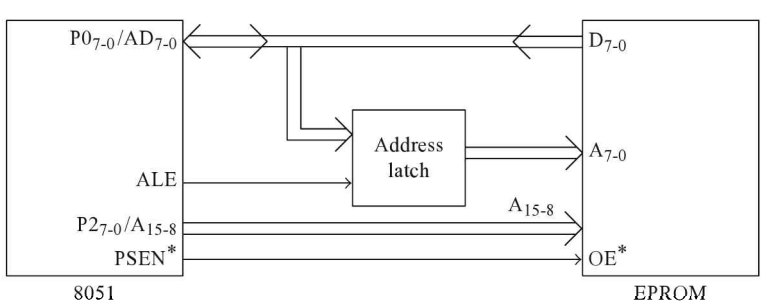


Fig. 29.5
Interfacing of external program memory

The external program memory address is always sent out as a 16-bit quantity, even if the amount of external memory is much less than 64K bytes. Thus whenever external program memory is used, P0 and P2 cannot be used for I/O purposes. The interesting aspect in the design of 8051 is that the instruction execution times do not depend on whether the instruction is in on-chip ROM or external EPROM.

■ 29.4 DATA MEMORY STRUCTURE

The 8051 has 128 bytes of internal on-chip RAM in the address range 00H–7FH for data storage. If more data memory is needed, it is necessary to use external data memory. The data memory structure of 8051 is shown in Fig. 29.6.

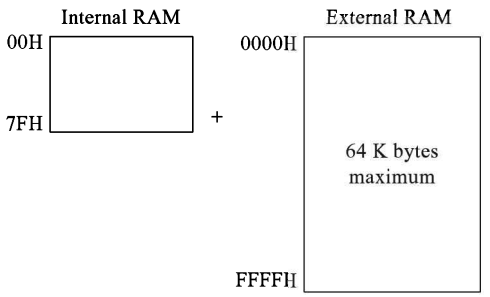


Fig. 29.6
Data memory structure

As can be seen from Fig. 29.6, it is possible to have an external data memory location with address as 50H, when there is already an internal RAM location with the same address 50H. Thus it can be concluded that the total amount of data memory can be a maximum of 64 K bytes excluding the internal data memory.

RD* and WR* signals output by 8051 on P3 pins are used for accessing the external data memory. External data memory address can be either 1- or 2-bytes wide. One-byte addresses are commonly used, but then only 256 bytes of external memory can be addressed. For most applications this should be adequate. In fact for many applications the internal program memory of 4 K bytes and internal data memory of 128 bytes should suffice. If only a little extra data memory is required, P0 pins can be used to provide an 8-bit address. Then upto 256 bytes of external data memory can be accessed. In such a case P2 is free for use as an I/O port, provided external program memory is not present.

If a little more than 256 bytes of external data memory are required, then a few lines of P2 can be used to page the external RAM. A scheme for providing 2K bytes of external data memory using eight pages of 256 bytes each is presented in Fig. 29.7.

In this scheme only three lines of P2 are used to select a page of 256 bytes. The remaining five lines of P2 can be used for I/O purposes. It is assumed that there is no external program memory. This is because if there is external program memory, 16 bits of address would have to be sent out, always on P0 and P2. In an instruction like 'MOVX A, @R1' (to be discussed later) only 8-bit address is sent out by 8051 on P0 pins, and it is the responsibility of the programmer to send out the three page bits using P2.

However, if a substantial amount of external data memory is required, then a full 16-bit address has to be sent out on P0 and P2. In such a case, the programmer is allowed to use instructions like

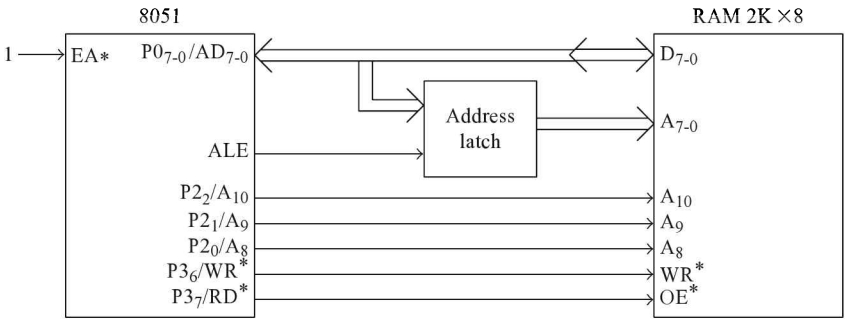


Fig. 29.7 Implementing 2K bytes of external RAM

'MOVX A, @DPTR' (the instruction will be explained later). External program memory and external data memory may be combined if desired by connecting RD* and PSEN* outputs of 8051 as inputs to an AND gate, and using the output of AND gate as the read strobe for the external program/data memory. This scheme is shown in Fig. 29.8. In such a case it is the responsibility of the programmer to ensure that program and data occupy non-overlapping memory locations.

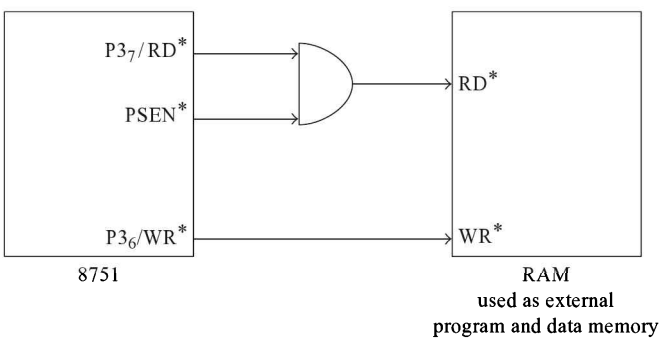


Fig. 29.8
Circuit for common external program and data memory

29.4.1 Internal RAM Organization

Internal data memory is addressed using 8 bits. Thus $2^8 = 256$ memory locations are possible, but 8051 contains only 128 bytes of internal data memory. They occupy the addresses 00H to 7FH. The address range 80H–FFH is allocated to accommodate a number of registers like A (accumulator), B register, DPTR (data pointer) and so on. These registers are called special function registers or SFRs. These SFRs are implemented in the internal RAM address space 80H to FFH, called the SFR space. Only 21 locations in the SFR space are used in 8051 for implementing the SFRs. The other locations in the SFR space do not exist physically. As such, the results will not be predictable if such addresses are used in an instruction. The two parts of the internal RAM are illustrated in Fig. 29.9.

29.4.2 Internal Data Memory Organization

The internal data memory of 128 bytes is divided into two groups—a set of eight registers and scratch pad memory. The eight registers are R0 to R7, and are implemented in the address range 00H–07H. The rest is scratch pad memory. In fact, 8051 provides the facility of four register banks, but only one

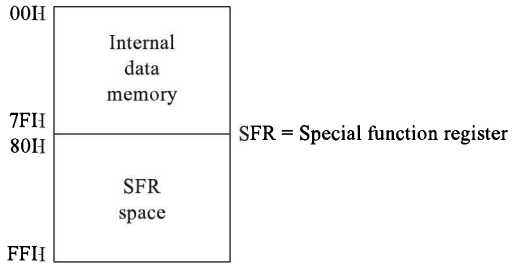


Fig. 29.9
Data memory and SFR space in internal RAM

bank of registers can be in use at any point of time. Two bits in PSW (program status word), which is an SFR, selects the register bank for use. The PSW will be described later. The register banks have the following addresses.

00H to 07H	Register bank 0;
08H to 0FH	Register bank 1;
10H to 17H	Register bank 2;
18H to 1FH	Register bank 3.

The availability of four register banks is a very useful feature for faster servicing of interrupts. The interrupted program may use one register bank, whereas the ISS may use another register bank. Hence there is no need to save and restore registers in the ISS, thus speeding up interrupt service. Even if there are nested interrupts this scheme can be used because there are four register banks.

If the user desires to use all the four register banks then only locations 20H to 7FH are available for scratch pad memory. Out of this range, memory locations 20H to 2FH can be used as bit-addressable RAM also in addition to being addressed as the usual byte locations. There are 16 bytes in this range for a total of $16 \times 8 = 128$ bits. These bits have the addresses in the range 00H to 7FH. The details are as provided in Fig. 29.10. These bits are addressed using 8 bits in an instruction like 'CLR 78H', which clears bit with address 78H (LS bit of location 2FH). Actually by using 8 bits it is possible to address 256-bit locations, but there are only 128 bits that are bit addressable in the internal data memory. Another 11 memory locations in the SFR area are also bit addressable. This constitutes another 88 bits that are bit addressable. They have bit addresses in the range 80H–FFH. These bit-addressable SFRs are described later. The remaining locations of the RAM in the range 30H–7FH can be used for storing variable data and implementing stack.

8051 stack: The stack in an 8051-based system is always implemented in the internal data memory. As the internal data memory range is only 00H to 7FH, the SP register is only 8 bits wide, and is one of the SFRs of 8051 with address 81H. If the user desires to use all the four register banks, and uses locations 20H to 2FH as bit-addressable memory, then the stack can start from location 30H. In such a case the SP is initialized with 2FH as is shown in Fig. 29.11.

Note that when SP contains 2FH, it means locations 2FH, 2EH, and so on upto 00H are having useful information and locations 30H, 31H, and so on upto 7FH are having useless information. Thus SP points to the highest addressed memory location having useful data. The SP is incremented by one before a push operation and is decremented by 1 after a pop operation. This is exactly the opposite of what happens to SP in 8085 microprocessor. The stack grows towards higher addresses, and variable data should be made to grow towards lower addresses. It is the programmer's responsibility to ensure that the stack and variable data do not overlap and destroy each other. As the stack is required to be implemented only in the internal data memory, only a small amount of stack is possible. This is one of the limitations of 8051.

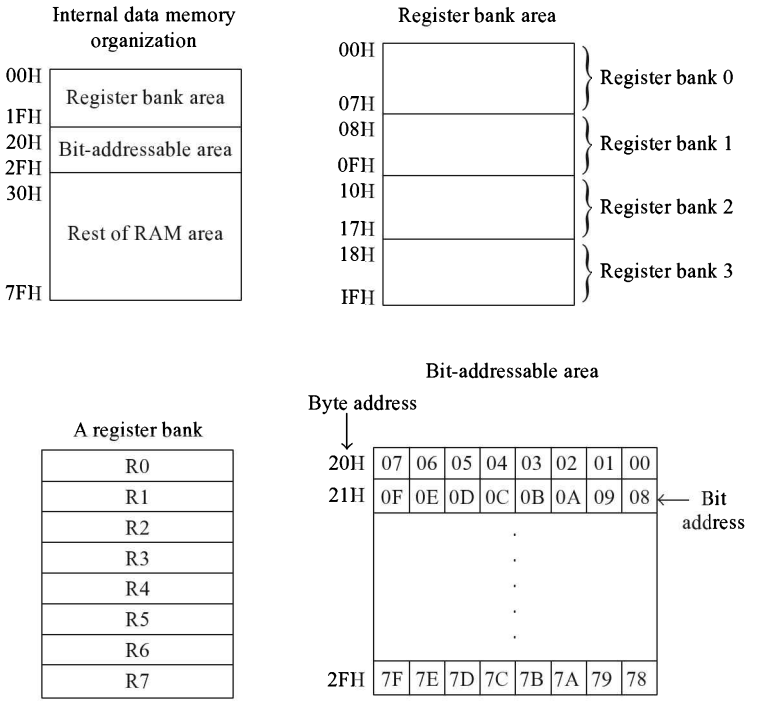


Fig. 29.10
Organization of internal data memory

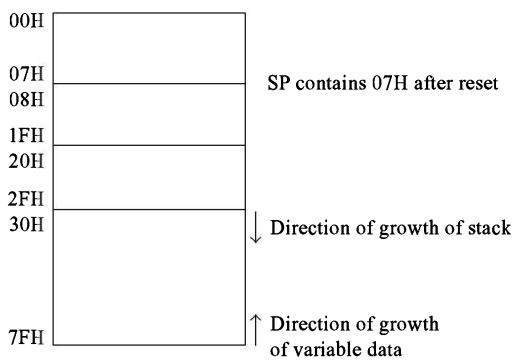


Fig. 29.11
Stack implementation in 8051

After reset of 8051, the SP is initialized to 07H. Thus locations 08H to 7FH can be used as stack area. In other words, it is assumed that only register bank 0 will be used and locations 20H to 2FH are not going to be used as bit-addressable area. However, if it is intended to use locations 20H to 2FH as bit-addressable area, then SP is initialized with 2FH.

29.4.3 SFR Area

The internal RAM of 8051 has two portions, as indicated in Fig. 29.9. Locations 00H to 7FH are used for register banks, bit-addressable memory locations, stack and variable data. This has been described in the previous section. Locations 80H to FFH is reserved for SFR area. Only 21 locations in the SFR

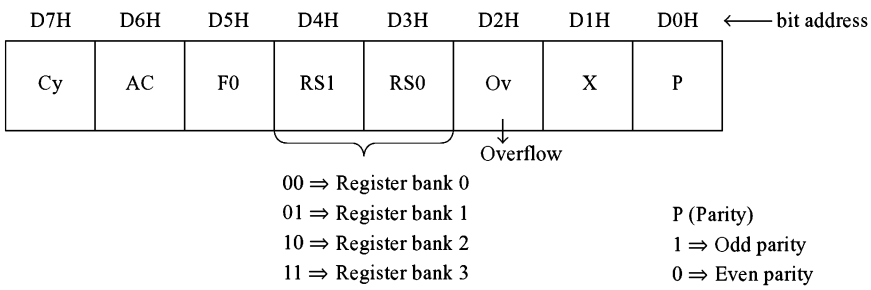
space are used in 8051 for implementing the SFRs. It is illegal to access other locations in the SFR space. Out of the 21 locations, 11 locations are bit-addressable SFR locations as are shown in Table 29.1. It may be noted that an SFR whose address has a 0H or 8H in the LS hex digit is a bit-addressable SFR.

Table 29.1 Bit-addressable SFRs

Sl. no.	*Register*	*Byte addr.*	*Bit addr.*
1	P0 (Port 0)	80H	80H to 87H
2	P1	90H	90H to 97H
3	P2	A0H	A0H to A7H
4	P3	B0H	B0H to B7H
5	PSW	D0H	D0H to D7H
6	A (Accumulator)	E0H	E0H to E7H
7	B	F0H	F0H to F7H
8	TCON (timer control)	88H	88H to 8FH
9	SCON (serial control)	98H	98H to 9FH
10	IE (interrupt enable)	A8H	A8H to AFH
11	IP (interrupt priority)	B8H	B8H to BFH

Only PSW register is described below. The other SFRs are explained later.

PSW register: The PSW register is nothing but the conventional flags register of a typical microprocessor. It is a bit-addressable SFR whose byte address is D0H. The bits in the PSW have the addresses D0H to D7H of which D7H is the address of the MS bit of PSW. Only 7 bits in the PSW are meaningful bits. The details of the PSW register are shown in Fig. 29.12.



P flag is affected based on A value immaterial of the type of instruction executed.
F0 is user-definable flag.
Cy flag is used as 1-bit accumulator in bit processing.

Fig. 29.12 Details of PSW register

The Cy flag has the usual meaning as in the case of 8085 processor. It is affected by the execution of add, subtract, and rotate through carry instructions. The Cy flag is also used as a 1-bit accumulator in bit manipulation instructions. Intel refers to bit manipulation as Boolean operations. Bit manipulation instructions are described later. The AC flag has the usual meaning as in the case of 8085 processor. The programmer does not use it, but the 8051 uses the value of this flag during the execution of DAA (decimal adjust accumulator). The V flag is useful in signed number arithmetic. If we are working with unsigned numbers there is no significance for this flag.

The P flag is set to 1 or cleared to 0 by the 8051 at the end of each instruction cycle depending on the number of 1s in the accumulator. The instruction executed need not be an arithmetic instruction.

If there are even number of 1s the P flag is cleared to 0. For odd number of 1s the P flag is set to 1. Note that this is exactly the opposite of the meaning of P flag in 8085 processor.

F0 is a general purpose flag bit, which is uncommitted and may be used as a general-purpose status flag. In fact, even bit D1H in the PSW can be used as a general-purpose flag. RS1 and RS0 bits in the PSW are used to select the current register bank as shown below.

RS1	*RS0*	*Selected register bank*
0	0	RB0 (register bank 0)
0	1	RB1
1	0	RB2
1	1	RB3

Note that Z flag and sign flags are not implemented in 8051. The F0 bit, for example, can be used as sign flag. The user will have to set this bit to 1 when the MS bit of result is 1. The Z flag is not much needed because the 8051 checks for a zero or non-zero condition by testing the contents of the accumulator directly.

Of the 21 SFR locations, ten contain byte information, which are not bit addressable, as that would not make any sense for these SFRs. These non-bit-addressable SFRs are shown as follows. These SFRs (excluding SP, which is already discussed) will be discussed later.

Sl. no.	*Register*	*Byte address*
1	SP (stack pointer)	81H
2	DPL (data pointer low)	82H
3	DPH (data pointer high)	83H
4	PCON (power control)	87H
5	TMOD (timer mode)	89H
6	TL0 (timer0 low)	8AH
7	TL1 (timer1 low)	8BH
8	TH0 (timer0 high)	8CH
9	TH1 (timer1 high)	8DH
10	SBUF (serial buffer)	99H

After reset of 8051, the SFRs will be initialized to the following values.

- SP contains 07H.
- Ports 0 to 3 will contain FFH.
- SBUF will have indeterminate (garbage) value.
- All other SFRs will have 00H.

■ 29.5 PROGRAMMER'S VIEW OF 8051

The programmer's view of 8051 is presented in Fig. 29.13.

From a programmer's viewpoint 8051 has the following.

- Eight registers R0 to R7 of 8-bit width;
  10 commonly used SFRs listed as follows;
- Registers A and B of 8-bit width;

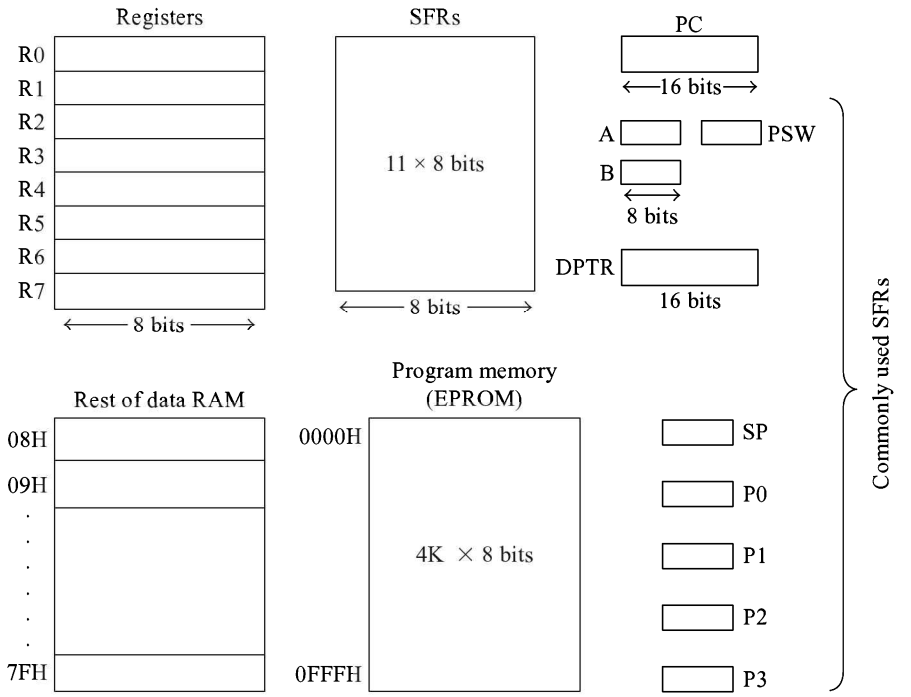


Fig. 29.13 Programmer's view of 8051

- PSW, SP, Ports P0, P1, P2 and P3-each of 8 bits;
- DPTR of 16-bit width comprising of DPH and DPL-each of 8 bits;
- Remaining 11 SFRs;
- Data RAM from 08H to 7FH;
- Program memory from 0000H to 0FFFH;
- Sixteen-bit PC.

29.6 ADDRESSING MODES OF 8051

The length of an 8051 instruction is 1, 2, or 3 bytes. However, except for a few instructions, the others are just 1- or 2-byte instructions. The opcode for an instruction is always 8 bits long. Hence $2^8 = 256$ opcodes are possible, of which 255 opcodes have been implemented. They can be classified into 111 types of instructions. If the crystal frequency is 12 MHz, then 64 instruction types are executed in just 1 μs, and the rest in just 2 μs (except for multiply and divide instructions that need 4 μs). Intel 8051 provides the following addressing modes for accessing data from internal data memory or external data memory.

- Immediate addressing;
- Register addressing;
- Direct addressing;

- Register indirect addressing;
- Indexed addressing;
- Implied addressing.

29.6.1 Immediate Addressing

In this type of addressing the source operand value is provided in the byte or bytes following the opcode, examples of which are shown in the following.

```
MOV A, #64H;          Moves 64H to A. 2-byte instruction.
MOV R6, #0E3H;        Moves E3H to R6. 2-byte instruction.
MOV DPTR, #1234H;     Moves 1234H to DPTR. 3-byte instruction.
```

The '#' symbol stands for immediate data. DPIR stands for Data Pointer. It is a 16-bit SFR. The DPTR points to an external data memory location. If a hexadecimal value starts with A to F, it should be preceded by 0, as in 0E3. The code for the 3-byte instruction 'MOV DPTR,#1234H' is stored in memory as '90 12 34', where 90H is the opcode for 'MOV DPTR'. It is to be noted that 12 34H is not stored in byte reversal form.

29.6.2 Register Addressing

In this type of addressing the source or destination operand is the contents of R*n* (*n* = 0 to 7). R*n* is specified in the opcode using 3 bits, examples of which are shown in the following.

```
MOV A, R3;      Moves contents of R3 to A. 1 byte instruction.
MOV R6, #30H;   Moves 30H to R6. 2 byte instruction.
MOV R5, A;      Moves contents of A to R5. 1 byte instruction.
```

Note that there is no instruction like 'MOV R5,R3'. The same effect is achieved using the instruction 'MOV R5,03H' or 'MOV 05H,R3'. This is because, in order to provide instructions like 'MOV R5,R3' an additional 64 opcodes are required, and already 255 out of the possible 256 opcodes are implemented. It should be noted that 'MOV R5,03' works fine in place of 'MOV R5,R3' only if the selected register bank is 0. If we are using register bank 1, 'MOV R5,0BH' has to be used for the same effect.

• 29.6.3 Direct Addressing

In this type of addressing, the source or destination operand address is specified using 8 bits in the instruction. Only internal data memory and SFRs can be directly addressed. In fact, SFRs are accessed using only direct addressing mode. External memory cannot be accessed using direct addressing mode in 8051. The following are some examples.

```
MOV 80H, R3 ; Moves contents of R3 to Port 0. P0 address is 80H.
MOV R6, 30H ; Moves contents of internal RAM location 30H to R6.
MOV R5, 03H ; Moves contents of R3 to R5. (Assuming Register bank is 0)
MOV 05H, R3 ; Moves contents of R3 to R5. (Assuming Register bank is 0)
```

Note that '80H' without '#' preceding it, refers to an address. All the above instructions are 2 bytes in length. Many assemblers allow 'MOV 80H,R3' to be written as 'MOV P0,R3'.

Advantage of SFR: There is a definite advantage in treating SFR as an internal RAM location instead of just treating it as a register. If a few more SFRs are added to the microcontroller in its improved version, then there is no need to add any more instruction codes to access these added SFRs. It is something like if the memory in the computer system is increased, there is no need to have additional instruction codes to access these added memory locations. To justify the above, it may be noted that 8752 is an improvement over 8051, as it has one more timer/counter. But the instruction set of 8051 and 8752 remain the same. Thus further (128−21) = 107 SFRs may be added to 8051 architecture without the need for additional instruction codes. On the other hand, if one more register, say G register, is added to 8085 architecture, then we must have additional opcodes for instructions like 'MOV A, G' and so on.

29.6.4 Register Indirect Addressing

In this type of addressing the source or destination operand address is provided in a register. In this mode, both internal and external data memory can be accessed. Only R0 and R1 can be used as 8-bit address registers. Only DPTR can be used as 16-bit address register. The following are some examples.

```
MOV 0F0H, @R1  ;Suppose (R1) = 40H. Then contents of internal RAM
               ;location 40H is moved to B. B address is F0H.
MOV @R0, 90H   ;Suppose (R0) = 30H. Then contents of Port 1 is moved
               ;to internal RAM location 30H. Port 1 has address 90H.
```

The '@' symbol stands for register indirect addressing. The above instructions are 2 bytes in length. Note that in the assembly language R0 to R7 and all the SFRs except A, and DPTR are expressed using their 8-bit SFR address. Exceptions to this are the push and pop instructions where even A, and DPTR are expressed using their 8-bit SFR address. However, many standard assemblers provide the facility of using the SFR name in assembly language programming, although in machine language the SFR address will be used (except for A, DPTR, and R0 to R7 in general). But some assemblers expect the SFR address, instead of SFR name, to be used in assembly language also.

Thus in many assemblers 'MOV 0F0H, @R1' can also be written as 'MOV B, @R1' which is more easily understandable. Similarly, 'MOV @R0, 90H' can be written as 'MOV @R0, P1'.

```
MOVX A, @R0    ;Suppose (R0) = 40H. Then contents of external RAM
               ;location 40H is moved to A.
MOVX @DPTR, A  ;Suppose (DPTR) = 1240H. Then contents of A is
               ;moved to external RAM location 1240H.
```

The 'X' in 'MOVX' refers to eXternal data memory. The above instructions are 1-byte long. Note that external data memory can be accessed using register indirect addressing only. Also only A register communicates with external memory.

29.6.5 Indexed Addressing

In this type of addressing the source operand is accessed from program memory only. It is intended for reading look-up tables present in program memory. The destination operand is always A register. Some examples follow.

```
MOVC A, @A+PC      ;Suppose (A) = 20H, (PC) = 1231H. Then contents
                   ;of program memory location 1251H is moved to A.

MOVC A, @A+DPTR    ;Suppose (A) = 20H, (DPTR) = 2230H. Then contents
                   ;of program memory location 2250H is moved to A.
```


The 'C' in 'MOVC' refers to code byte. The above instructions are 1 byte in length.

29.6.6 Implied Addressing

In this type of addressing there will be a single operand, and this happens to be a specific register only, like the accumulator. Intel refers to this type of addressing as 'register specific instructions'. The following are some examples.

```
RL A   ;Rotate left A. Only A register can be rotated left / right.
SWAP A ;Swap nibbles in A. Swapping possible with A register only.
```

The above instructions are 1-byte in length.

■ 29.7 INSTRUCTION SET OF 8051

The instructions of 8051 can be classified into the following groups.

- Data transfer group;
- Arithmetic group;
- Logical group;
- Program branch group;
- Bit-processing group (Intel calls it Boolean variable manipulation).

In an instruction that has two operands, the first operand specifies the destination and the second specifies the source. This is the same as in the Intel 8085 microprocessor.

29.7.1 Data Transfer Group

There are 28 instruction types in the data transfer type of operations. They account for 79 opcodes, which are indicated in Table 29.2. The table indicates the mnemonic, length of the instruction, execution time in terms of machine cycles, and number of opcodes for the type of instruction. Thus, if the execution time is shown as one machine cycle, it takes 1 μs for execution if the crystal frequency is 12 MHz. None of the flags are affected by the execution of data transfer instructions. However, if A value is altered because of the execution of a data transfer instruction, the P flag is set to 1 if there are odd number of 1s, else it is reset to 0. Also if data is moved to PSW register all the flags are obviously affected. In the description of instructions, the following convention is used.

Rn = any of the registers R0 to R7;
Ri = any of the registers R0 and R1;
d8 = any 8-bit immediate data (range 00H–FFH);

Table 29.2 Data Transfer Group Instructions

Sl. no.	*Mnemonic*	*No. of bytes*	*Exec. time*	*Opcodes*
1.	MOV A, Rn	1	1	8
2.	MOV A, a8	2	1	1
3.	MOV A, @Ri	1	1	2
4.	MOV A, #d8	2	1	1
5.	MOV Rn, A	1	1	8
6.	MOV Rn, a8	2	2	8
7.	MOV Rn, #d8	2	1	8
8.	MOV a8, A	2	1	1
9.	MOV a8, Rn	2	2	8
10.	MOV a8, a8	3	2	1
11.	MOV a8, @Ri	2	2	2
12.	MOV a8, #d8	3	2	1
13.	MOV @Ri, A	1	1	2
14.	MOV @Ri, a8	2	2	2
15.	MOV @Ri, #d8	2	1	2
16.	MOV DPTR, #d16	3	2	1
17.	MOVC A, @A+DPTR	1	2	1
18.	MOVC A, @A+PC	1	2	1
19.	MOVX A, @Ri	1	2	2
20.	MOVX A, @DPTR	1	2	1
21.	MOVX @Ri, A	1	2	2
22.	MOVX @DPTR, A	1	2	1
23.	PUSH a8	2	2	1
24.	POP a8	2	2	1
25.	XCH A, Rn	1	1	8
26.	XCH A, a8	2	1	1
27.	XCH A, @Ri	1	1	2
28.	XCHD A, @Ri	1	1	2

d16 = any 16-bit immediate data (range 0000H–FFFFH);
a8 = 8-bit address. This could refer to an SFR or a data RAM location.

Examples

'MOV R3, 25H' is a typical example for the general type of instruction 'MOV Rn, a8'. The contents of internal RAM location 25H are moved to R3. It is a 2-byte instruction, and the execution time is two machine cycles.

'MOV 0F0H, #25H' is a typical example for the general type of instruction 'MOV a8, #d8'. The immediate data 25H is moved to register B, which is an SFR with address F0H. It is a 3-byte instruction, and the execution time is two machine cycles. The instruction can also be written as 'MOV B, #25H'.

'PUSH 0E0H' is a typical example for the general type of instruction 'PUSH a8'. The contents of A register (SFR address E0H) are pushed on the stack. The SP is incremented by 1 in this operation. It is a 2-byte instruction, and the execution time is two machine cycles. The instruction can also be written as 'PUSH A'.

'POP 30H' is a typical example for the general type of instruction 'POP a8'. The contents of stack top is moved to internal RAM location 30H. The SP is then decremented by 1. It is a 2-byte instruction, and the execution time is two machine cycles.

'XCH A, R3' is a typical example for the general type of instruction 'XCH A, Rn'. The contents of A and R3 are exchanged. It is a 1-byte instruction, and the execution time is one machine cycle.

'XCHD A, @R1' is a typical example for the general type of instruction 'XCHD A, @Ri'. The contents of LS digit of A and LS digit of internal RAM location pointed by R1 are exchanged. If A contents are 35H, R1 contains 45H, and internal RAM location 45H contains 67H, then A contents are changed to 37H and internal RAM location 45H contents are changed to 65H. It is a 1-byte instruction, and the execution time is one machine cycle. The effect of execution of this instruction is represented as shown below.

	Before	*After*
(A)	35H	37H
(R1)	45H	
(45H)	67H	65H

29.7.2 Arithmetic Group

There are 24 instruction types in the arithmetic group of instructions, which account for 64 opcodes. They are indicated in Table 29.3. The Cy, AC, and V flags are affected based on the result in the case of ADD, ADDC, and SUBB instructions. The multiply and divide instructions clear the Cy flag, and do not affect the AC flag. The V flag is set to 1 after multiply instruction, if the result is greater than FFH, else it is cleared. Similarly after divide instruction, the V flag is set to 1 if the B content was 00 before division, else it is cleared. DA A instruction affects the Cy flag only. Other instructions do not affect any flags. However, when A value is altered because of execution of an instruction, the P flag is affected.

Table 29.3 Arithmetic Group of Instructions

Sl. no.	*Mnemonic*	*No. of bytes*	*Exec. time*	*Opcodes*
1.	ADD A, Rn	1	1	8
2.	ADD A, a8	2	1	1
3.	ADD A, @Ri	1	1	2
4.	ADD A, #d8	2	1	1
5.	ADDC A, Rn	1	1	8
6.	ADDC A, a8	2	1	1
7.	ADDC A, @Ri	1	1	2
8.	ADDC A, #d8	2	1	1
9.	SUBB A, Rn	1	1	8
10.	SUBB A, a8	2	1	1
11.	SUBB A, @Ri	1	1	2
12.	SUBB A, #d8	2	1	1
13.	INC A	1	1	1
14.	INC Rn	1	1	8
15.	INC a8	2	1	1
16.	INC @Ri	1	1	2
17.	DEC A	1	1	1
18.	DEC Rn	1	1	8
19.	DEC a8	2	1	1
20.	DEC @Ri	1	1	2
21.	INC DPTR	1	2	1
22.	MUL AB	1	4	1
23.	DIV AB	1	4	1
24.	DA A	1	1	1

Examples

'ADD A, #25H' is a typical example for the general type of instruction 'ADD A, #d8'. The immediate data 25H is added to A, and the result is stored in A. It is a 2-byte instruction, and the execution time is one machine cycle. Note that for add and subtract instructions the result is always stored in A.

'ADDC A, @R0' is a typical example for the general type of instruction 'ADDC A, @Ri'. The contents of internal RAM location pointed by R0 is added to A. It is a 1-byte instruction, and the execution time is one machine cycle.

'SUBB A, R3' is a typical example for the general type of instruction 'SUBB A, Rn'. SUBB stands for subtract with borrow. The contents of R3 is subtracted from A contents along with borrow, and the result is stored in A. 'SUBB A, R3' is a 1-byte instruction, and the execution time is one machine cycle. Note that there is no 'SUB A, R3' which subtracts R3 contents from A. This was due to paucity of opcodes. If this operation is desired, then it is necessary to execute 'CLR C' and 'SUBB A, R3'. 'CLR C' clears the Cy flag to 0.

'INC 30H' is a typical example for the general type of instruction 'INC a8'. The contents of internal RAM location 30H is incremented by one. It is a 2-byte instruction, and the execution time is one machine cycle.

'INC DPTR' increment contents of DPTR by one. It is a 1-byte instruction, and the execution time is two machine cycles. 'DEC DPTR' instruction is strangely not implemented in 8051.

'MUL AB' multiplies the contents of A and B registers, treating them as unsigned numbers. The 16-bit result is stored in B (MS byte) and A (LS byte). If the result is larger than FFH, then the V flag is set to one. The Cy flag is always cleared after the multiply instruction. It is a 1-byte instruction that needs four machine cycles for execution.

'DIV AB' divides the contents of A register by the contents of B register, treating them as unsigned numbers. The 8-bit quotient is stored in A and the 8-bit remainder in B register. The V flag is set to 1 only if B content is 00H before division. Also A and B values are undefined after divide operation. The Cy flag is always cleared after the divide instruction. It is a 1-byte instruction that needs four machine cycles for execution.

29.7.3 Logical Group

There are 25 instruction types in the logical group of instructions. They account for 49 opcodes which are indicated in Table 29.4. Only RRC and RLC instructions affect the Cy flag. The other instructions do not affect any flag.

Examples

'ANL A,R3' is a typical example for the general type of instruction 'ANL A,Rn'. The contents of R3 is ANDed with A contents and result stored in A. It is a 1-byte instruction, and the execution time is one machine cycle. This is similar to 'ANA' instruction of Intel 8085. Similarly OR and Ex-OR operations can also be performed. Note that in these operations the destination operand is always A register.

'CLR A' instruction results in A contents becoming 00H. It is not possible to clear the contents of any other register or memory location. It is a 1-byte instruction, and the execution time is one machine cycle.

'CPL A' instruction results in complementing every bit of A register. It is not possible to complement the contents of any other register or memory location. It is a 1-byte instruction, and the execution time is one machine cycle.

'RL A' rotates left contents of A by 1-bit position. It is similar to 'RLC' instruction of Intel 8085, except that none of the flags are affected. Contents of no other register or memory location can be rotated left or right. 'RL A' is a 1-byte instruction that is executed in one machine cycle.

Table 29.4 Logical Group of Instructions

Sl. no.	*Mnemonic*	*No. of bytes*	*Exec. time*	*Opcodes*
1.	ANL A, Rn	1	1	8
2.	ANL A, a8	2	1	1
3.	ANL A, @Ri	1	1	2
4.	ANL A, #d8	2	1	1
5.	ANL a8, A	2	1	1
6.	ANL a8, #d8	3	2	1
7.	ORL A, Rn	1	1	8
8.	ORL A, a8	2	1	1
9.	ORL A, @Ri	1	1	2
10.	ORL A, #d8	2	1	1
11.	ORL a8, A	2	1	1
12.	ORL a8, #d8	3	2	1
13.	XRL A, Rn	1	1	8
14.	XRL A, a8	2	1	1
15.	XRL A, @Ri	1	1	2
16.	XRL A, #d8	2	1	1
17.	XRL a8, A	2	1	1
18.	XRL a8, #d8	3	2	1
19.	CLR A	1	1	1
20.	CPL A	1	1	1
21.	RL A	1	1	1
22.	RLC A	1	1	1
23.	RR A	1	1	1
24.	RRC A	1	1	1
25.	SWAP A	1	1	1

'RLC A' rotates left contents of A together with Cy flag by 1-bit position. It is the same as the 'RAL' instruction of Intel 8085. Only Cy flag is affected. 'RLC A' is a 1-byte instruction that is executed in one machine cycle.

'RR A' rotates right contents of A by 1-bit position. It is similar to 'RRC' instruction of Intel 8085, except that none of the flags are affected. 'RR A' is a 1-byte instruction that is executed in one machine cycle.

'RRC A' rotates right contents of A together with Cy flag by 1-bit position. It is the same as the 'RAR' instruction of Intel 8085. Only Cy flag is affected. 'RRC A' is a 1-byte instruction that is executed in one machine cycle.

'SWAP A' instruction exchanges the LS hex digit with the MS hex digit in A. It is functionally the same as executing RLA/RRA four times. The flags are not affected and it is a 1-byte instruction that is executed in one machine cycle.

29.7.4 Bit-Processing Group

There are 17 instruction types that come under bit processing of instructions. They account for 17 opcodes. The carry bit acts like 'single-bit accumulator' in many bit-processing instructions. The bit-processing instructions are indicated in Table 29.5. These instructions do not affect any flags, however, instructions like 'ANL C, bit' will obviously affect the Cy flag. In the description of these instructions, the following convention is used.

Table 29.5 Bit Processing Instructions

Sl. no.	*Mnemonic*	*No. of bytes*	*Exec. time*	*Opcodes*
1.	CLR C	1	1	1
2.	CLR bit	2	1	1
3.	SETB C	1	1	1
4.	SETB bit	2	1	1
5.	CPL C	1	1	1
6.	CPL bit	2	1	1
7.	ANL C, bit	2	2	1
8.	ANL C, /bit	2	2	1
9.	ORL C, bit	2	2	1
10.	ORL C, /bit	2	2	1
11.	MOV C, bit	2	1	1
12.	MOV bit, C	2	2	1
13.	JC rel	2	2	1
14.	JNC rel	2	2	1
15.	JB bit, rel	3	2	1
16.	JNB bit,rel	3	2	1
17.	JBC bit,rel	3	2	1

bit = 8-bit address of a bit which is bit addressable;
rel = 8-bit signed displacement (range −128 to +127) relative to first byte of the following instruction.

Examples

'CLR C' instruction results in Cy flag becoming 0. It is a 1-byte instruction, and the execution time is one machine cycle.

'SETB 0D5H' is a typical example for the general type of instruction 'SETB bit'. It sets to 1 bit D5 (which is F0 flag in PSW). It is a 2-byte instruction, and the execution time is one machine cycle. The instruction could also be written as 'SETB PSW.5' or 'SETB F0'.

'CPL C' instruction results in complementing C bit. Similarly using 'CPL bit' instruction a bit can be complemented.

'ANL C, 07' is a typical example for the general type of instruction 'ANL C, bit'. It ANDs C bit with bit 7 (which is MS bit in internal RAM location 20H) and stores the result in C. It is a 2-byte instruction, and the execution time is two machine cycles.

'ORL C, /07' is a typical example for the general type of instruction 'ORL C, /bit'. It ORs C bit with complement of bit 7 (bit 7 is MS bit in internal RAM location 20H) and stores the result in C. It is a 2-byte instruction, and the execution time is two machine cycles.

'MOV C, 07' is a typical example for the general type of instruction 'MOV C, bit'. It moves the contents of bit 7 (which is MS bit in internal RAM location 20H) to C. It is a 2-byte instruction, and the execution time is one machine cycle.

Thus if it is required to set the Cy flag if and only if LS bit of P0 is 1, MS bit of A register is 1, and V flag is 0, the following instructions are to be executed.

```
MOV C, 80H    ;80H is address of LS bit of Port 0
ANL C, 0E7H   ;E7H is address of MS bit of A register
ANL C,/0D2H   ;D2H is address of Overflow flag in PSW
```

These instructions could have been written as

```
MOV C, P0.0
ANL C, ACC.7
ANL C, /OV ; or ANL C, /PSW.2
```

'JC LOOP' is a typical example for the general type of instruction 'JC rel'. If the Cy flag is set to 1, then a branch to location LOOP takes place. If Cy flag = 0, execution proceeds with the next instruction. LOOP is a symbolic location, but the assembler generates an 8-bit signed displacement from the instruction following 'JC LOOP'. Thus branch can take place to a location which is 128 bytes behind or 127 bytes ahead of the instruction following JC. It is a 2-byte instruction, and the execution time is two machine cycles.

'JNB 80H, BACK' is a typical example for the general type of instruction 'JNB bit, rel'. If LS bit of Port 0 (bit address 80H) is reset to 0, then a branch to location BACK takes place. If this bit is set to 1, execution proceeds with the next instruction. BACK is a symbolic location, but the assembler generates an 8-bit signed displacement from the instruction following 'JNB 80H,BACK'. It is a 3-byte instruction, and the execution time is two machine cycles. The instruction could also be written as 'JNB P0.0, BACK'.

'JBC 80H, BACK' is a typical example for the general type of instruction 'JBC bit, rel', where JBC stands for 'jump if bit is set and then clear bit'. If LS bit of P0 (bit address 80H) is set to 1, then a branch to location BACK takes place and the bit will be cleared to 0. If this bit was reset to 0, execution proceeds with the next instruction. BACK is a symbolic location, but the assembler generates an 8-bit signed displacement from the instruction following 'JNB 80H, BACK'. It is a 3-byte instruction, and the execution time is two machine cycles. The instruction could also be written as 'JBC P0.0, BACK'. In reality, instructions like JC, JNB, and JBC could be discussed under program branch instructions.

29.7.5 Program Branch Group

There are 17 instruction types that come under program branch instructions. They account for 46 opcodes. The program branch instructions are indicated in Table 29.6. Only CJNE instruction affects the Cy flag, while other instructions do not affect any flag. In the description of these instructions, the following convention is used.

addr11 = 11-bit address
addr16 = 16-bit address

Examples

'LJMP LOOP' is a typical example for the general type of instruction 'LJMP addr16'. LJMP stands for long jump. It performs a branch to symbolic location LOOP. The assembler generates a 16-bit address corresponding to location LOOP. A branch to any location in program memory is possible using LJMP instruction. It is a 3-byte instruction, and the execution time is two machine cycles.

'ACALL SUBR' is a typical example for the general type of instruction 'ACALL addr11'. It is a 2-byte instruction, and the execution time is two machine cycles. ACALL stands for absolute call. It causes a branch to subroutine at symbolic location SUBR after saving address of the next instruction on the stack top. For the purpose of ACALL and AJMP instructions, the program memory can be thought of as being divided into the following blocks of 2K bytes—0000H to 07FFH, 0800H to

Table 29.6 Program Branch Instructions

Sl. no.	*Mnemonic*	*No. of bytes*	*Exec. time*	*Opcodes*
1.	ACALL addr11	2	2	8
2.	LCALL addr16	3	2	1
3.	RET	1	2	1
4.	RETI	1	2	1
5.	AJMP addr11	2	2	8
6.	LJMP addr16	3	2	1
7.	SJMP rel	2	2	1
8.	JMP @A+DPTR	1	2	1
9.	JZ rel	2	2	1
10.	JNZ rel	2	2	1
11.	CJNE A,a8,rel	3	2	1
12.	CJNE A,#d8,rel	3	2	1
13.	CJNE Rn,#d8,rel	3	2	8
14.	CJNE @Ri,#d8,rel	3	2	2
15.	DJNZ Rn,rel	2	2	8
16.	DJNZ a8,rel	3	2	1
17.	NOP	1	1	1

0FFFH, and so on, and finally F800H to FFFFH. As an example, if the address of the instruction after ACALL is in the block 1800H to 1FFFH, then the starting address of the subroutine using ACALL is restricted to be within this block of 1800H to 1FFFH. For any address within such a block, the MS 5 bits remain the same. As an example, the MS 5 bits of 1800H and 1FFFH are 00011. This 00011 is called the block address. It provides MS 5 bits of subroutine address. Hence just 11 bits of address is enough to specify the address of subroutine within the block.

Further, the block of 2K = 0800H bytes of memory can be thought of as being divided into eight pages each of 256 = 100H bytes. Hence to select a location in a block of 2K bytes, we have to select a page within the block using 3 bits, and a location within the page using 8 bits. The ACALL instruction is implemented using only 2 bytes. The first byte will be PPP 10001, where PPP indicates page number within the block, and 10001 is the code for ACALL. The second byte contains the address within the page.

As an example, consider 'ACALL SUBR' being at location 1834H in external program memory. If SUBR is location 1C85H, 'ACALL SUBR' is coded as follows. First byte will be 100 10001B = 91H, where 100 indicates select page 4 within the block and 10001 is the opcode for ACALL. The second byte will be 85H that provides the LS byte of subroutine address.

The MS 5 bits of subroutine address will be the block address. This is 00011 as described earlier. The next 3 bits provide page address. This will be 100 to indicate page 4 within block 00011. Thus, branch takes place to the subroutine at address 00011 100 85H = 1C85H. If SUBR is at location 1D85H, then 'ACALL SUBR' is coded as follows. First byte will be 101 10001B = B1H, where 101 indicates select page 5 within the block and 10001 is the opcode for ACALL. The second byte will be 85H that provides the LS byte of subroutine address.

The MS 5 bits of subroutine address will be 00011 that gives the block address. The next 3 bits will be 101 to indicate page 5. Thus, branch takes place to the subroutine at address 00011 101 85H = 1D85H. From the above it should be clear that there are eight opcodes for ACALL instruction depending on the page number within the block where the subroutine starts. As can be seen, two of the eight possible opcodes for ACALL are 91H and B1H.

'SJMP LOOP' is a typical example for the general type of instruction 'SJMP rel'. SJMP stands for short jump. 'SJMP LOOP' causes a branch to location LOOP. LOOP is a symbolic location, but the

assembler generates an 8-bit signed displacement from the instruction following 'SJMP LOOP'. It is a 2-byte instruction, and the execution time is two machine cycles. In the instruction set of 8051 there is no HALT instruction. Thus, to terminate a program 'LOOP: SJMP LOOP' is used.

It can be noted from Table 29.6 that there are three kinds of unconditional jumps—SJMP, AJMP, and LJMP. However, JMP is a generic name that can be used if the programmer does not care which way the jump is encoded. However, some assemblers expect atleast one of SJMP, AJMP, or LJMP to be used. Similarly, it can be noted that there are two kinds of unconditional calls—ACALL and LCALL. However, CALL is a generic name that can be used if the programmer does not care which way the call is encoded. However, some assemblers expect ACALL or LCALL to be used. There is no SCALL instruction in 8051.

'JZ BACK' is a typical example for the general type of instruction 'JZ rel'. If A register contents is 00H, then a branch to location BACK takes place. If A register contents are unequal to 00H, execution proceeds with the next instruction. BACK is a symbolic location, but the assembler generates an 8-bit signed displacement from the instruction following 'JZ BACK'. It is a 2-byte instruction, and the execution time is two machine cycles. It is to be noted that JZ instruction tests the contents of A register for a 00H value, and there is no Z flag in 8051.

The RET instruction functions as pop from stack top to PC. It is similar to the RET instruction of 8085 (except that after the RET instruction of 8051, the SP value is decremented by 2). It is an 1-byte instruction that is executed in two machine cycles.

The RETI instruction also functions as pop from stack top to PC. In addition, it restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. RETI stands for return from interrupt. An interrupt service routine ends with RETI, and an ordinary subroutine ends with RET instruction. RETI is a 1-byte instruction that is executed in two machine cycles. The use of RETI would become clear after 8051 interrupt system is described in detail later.

'DJNZ 80H, BACK' is a typical example for the general type of instruction 'DJNZ a8, rel'. DJNZ stands for 'decrement and jump if not zero'. The P0 (address 80H) contents are decremented by 1. If it is not 00H after the decrement, then a branch to location BACK takes place. Otherwise execution proceeds with the next instruction. BACK is a symbolic location, but the assembler generates an 8-bit signed displacement from the instruction following 'DJNZ 80H, BACK'. It is a 3-byte instruction, and the execution time is two machine cycles. The instruction could also be written as 'DJNZ P0, BACK'.

'CJNE R3,#80H,BACK' is a typical example for the general type of instruction 'CJNE Rn,#d8,rel'. CJNE stands for 'compare and jump if not equal'. The CJNE instruction combines compare and jump into a single compact instruction. In this case, R3 contents are compared with 80H. If R3 contents are different from 80H then a branch to location BACK takes place. Otherwise execution proceeds with the next instruction. BACK is a symbolic location, but the assembler generates an 8-bit signed displacement from the instruction following 'CJNE R3,#80H, BACK'. It is a 3-byte instruction, and the execution time is two machine cycles. However, 8051 does not have instructions like 'compare and jump if above' or 'compare and jump if below'.

■ 29.8 PROGRAMMING EXAMPLES

The programming examples provided here are only to familiarize the reader with the instruction set of 8051, especially with those that are new to 8051 compared to 8085. These programs were tested on SDA 31 kit manufactured by M/s Advanced Electronic Systems, Bangalore—560 055.

29.8.1 Shift a Multi-byte BCD Number to the Right

Write an 8051 assembly language program to shift an 8-digit BCD number by two digits to the right. The 8-digit BCD number is assumed to be in internal data memory locations 20H, 21H, 22H, and 23H. Location 20H is having the MS byte.

For the trace that follows, it is assumed that the BCD number is 12 34 56 78, with 12 at location 20H. After the program is executed, contents of location 20H is cleared to 00H, and the LS byte shifted out will be available in A.

```
        CLR A      ;    (A) = 00
        XCH A,20H  ; (20H) = 00, (A) = 12,
        XCH A,21H  ; (21H) = 12, (A) = 34,
        XCH A,22H  ; (22H) = 34, (A) = 56,
        XCH A,23H  ; (23H) = 56, (A) = 78,
STOP:   SJMP STOP
```

29.8.2 Binary to BCD Conversion

Write an 8051 assembly language program to convert an 8-bit binary number to its equivalent BCD value. The 8-bit binary number is at external RAM location 30H. The result is to be stored in locations 31H and 32H, with 31H having the MS portion of result.

In the trace that follows it is assumed that the data at location 30H is FEH. The program converts binary value FEH to BCD value of 0254, and stores it in locations 31H and 32H.

```
        MOV P2, #00H     ;Set MS byte of external data address as 00
        MOV R0, #30H     ;Load R0 with address of data

        MOVX A, @R0      ;(A) = FEH (254 decimal)
        MOV B, #0AH      ;(B) = 0AH (10 decimal)

        DIV AB           ;(A) = 19H (25 decimal), (B) = 04
        MOV R2, B        ;(R2) = 04

        MOV B, #0AH      ;(B) = 0AH
        DIV AB           ;(A) = 02, (B) = 05

        INC R0           ;(R0) = 31H
        MOVX @R0, A      ;(0031) = 02 (Store MS portion of result)

        MOV A, B         ;(A) = 05
        SWAP A           ;(A) = 50
        ADD A, R2        ;(A) = 54

        INC R0           ;(R0) = 32H
        MOVX @R0, A      ;(0032) = 54 (Store LS portion of result)
EXIT:   SJMP EXIT        ;Stop
```

• 29.8.3a BCD to Binary Conversion

Write an 8051 assembly language program to convert a two-digit BCD number to its equivalent binary value. The two-digit BCD number is at external RAM location 200H. The result is to be stored in location 201H.

In the program that follows, two software up counters are made use of. They start from 00 value initially. One counter counts up in decimal using A register, while the other counter counts up in binary using R2 register. When the decimal up counter reaches the desired BCD value, the count up is stopped. At that point the binary counter will have the equivalent binary value.

In the trace that follows it is assumed that the data at location 200H is 15. The program converts BCD value 15 to binary value of 0FH, and stores it in location 0201H.

```
        MOV DPTR, #0200H   ;(DPTR) = 0200H
        MOVX A, @DPTR      ;(A) = 15
        MOV R3, A          ;(R3) = 15

        CLR A              ;(A) = 00
        MOV R2, #00        ;(R2) = 00

LOOP:   ADD A, #01         ;(A) = 01, 02, .., 09, 0A, 11, .., 15
        DAA                ;(A) = 01, 02, .., 09, 10, 11, .., 15
        INC R2             ;(R2)= 01, 02, .., 09, 0A, 0B, .., 0F
        CJNE A,03H,LOOP    ;Repeat till A value becomes same as R3

        INC DPTR           ;(DPTR) = 0201H
        MOV A, R2          ;(A) = 0FH
        MOVX @DPTR, A      ;(0201) = 0FH
EXIT:   SJMP EXIT          ;Halt
```

29.8.3b BCD to Binary Conversion (Alternative)

Write an 8051 assembly language program to convert a two-digit BCD number to its equivalent binary value. The two-digit BCD number is at external RAM location 200H. The result is to be stored in location 201H. In the program shown as follows, the MS digit of the BCD number is multiplied by 10 (0AH). For example, if the BCD number is 95, 9 is multiplied by 0AH, to get 5AH. Then the LS digit of the BCD number, which is five, is added to get 5FH which is the binary equivalent of the BCD number 95.

In the following trace it is assumed that the data at location 200H is 95. The program converts BCD value 95 to binary value of 5FH, and stores it in location 0201H.

```
        MOV DPTR, #0200H   ; (DPTR) = 0200H
        MOVX A, @DPTR      ; (A) = 95
        MOV R2, A          ; (R2) = 95

        SWAP A             ; (A) = 59
        ANL A, #0FH        ; (A) = 09H

        MOV B, #0AH        ; (B) = 0AH
        MUL AB             ; (A) = 5AH, (B) = 00H

        ANL R2, #0FH       ; (R2) = 05H
        ADD A, R2          ; (A) = 5FH

        INC DPTR           ; (DPTR) = 0201H
        MOVX @DPTR, A      ; (0201) = 5FH
HALT:   SJMP HALT          ; Stop
```

29.8.4 Hex to ASCII Conversion

Write an 8051 assembly language program to convert a single-digit hexadecimal number to its equivalent ASCII value. The single-digit hexadecimal number is at external RAM location 300H. The result is to be stored in location 301H.

In the following trace it is assumed that the data at location 300H is 00. The program converts hexadecimal value 00 to ASCII value of 30H, and stores it in location 0301H. The program uses look-up table approach. Immediately after the RET instruction, the look-up table containing the ASCII codes for '0' to 'F' is stored.

In the CONVRT subroutine when 'MOVC A, @A+PC' is executed, the PC value taken for calculation is the address of RET instruction. So to move past the 1-byte RET instruction and fetch result from look up table, A value is incremented by 1 in the main program before branching to CONVRT subroutine.

```
        MOV DPTR, #0300H  ;(DPTR) = 0300H
        MOVX A, @DPTR     ;(A) = 00H
        INC A             ;(A) = 01H

        ACALL CONVRT      ;(A) = 30H

        INC DPTR          ;(DPTR) = 0301H
        MOVX @DPTR, A     ;(0301) = 30H
STOP:   SJMP STOP         ;Halt

CONVRT: MOVC A, @A+PC     ;(A) = 30H
        RET               ;Return to main program

        30H ;ASCII code for '0'
        31H
        32H
        33H
        34H
        35H
        36H
        37H
        38H
        39H ; ASCII code for '9'
        41H ; ASCII code for 'A'
        42H
        43H
        44H
        45H
        46H ; ASCII code for 'F'
```

29.8.5 Bit Manipulation Program

Write an 8051 assembly language program to modify a given byte at internal RAM location 40H as detailed below, and store the modified byte at internal RAM location 41H.

Bit 0 is to be set to 1;
Bit 2 is to be reset to 0;
Bit 4 is to be complemented;
Bit 6 value should become the same as bit 5;
Bit 7 value should become AND of bit 1 and complement of bit 3.

In the trace that follows, it is assumed that the data at internal RAM location 40H is 25H. The program modifies it to 71H and stores it in internal RAM location 41H.

In the program that follows, contents of internal RAM location 40H is moved to internal RAM location 20H, which is a bit-addressable RAM location. This is done in order to carry out bit manipulation on this byte. In this program 'MOV C, 20.5' stands for 'move to Cy flag bit 5 of byte at internal RAM location 20H'. It could as well have been written as 'MOV C, 5'.

```
        MOV 20H, 40H ;(20H) = 0010 0101

        SETB 20.0    ;(20H) = 0010 0101
        CLR 20.2     ;(20H) = 0010 0001
        CPL 20.4     ;(20H) = 0011 0001

        MOV C, 20.5  ;(C) = 1. It can be written as 'MOV C, 5' also.
        MOV 20.6,C   ;(20H) = 0111 0001

        MOV C, 1     ;(C) =0. It can be written as 'MOV C,20.1' also.
        ANL C, /3    ;(C) = (0 AND 1) = 0
        MOV 7, C     ;(20H) = 0111 0001 = 71H

        MOV 41H, 20H ;(41H) = 71H
EXIT:   SJMP EXIT
```

29.8.6 Conversion of Four-digit Hex to ASCII

Write an 8051 assembly language program to convert a four-digit hexadecimal number to equivalent ASCII. The four-digit hex number is at internal RAM locations 30H and 31H. The equivalent ASCII is to be stored in four internal RAM locations starting from 50H.

```
        MOV R0, #30H ;(R0) = 30H
        MOV R1, #40H ;(R1) = 40H
        MOV R2, #02H ;(R2) = 02H. R2 is used as loop counter.

;The 'REPT' loop converts a 4 digit hexadecimal number, say,
;12CD at internal RAM locations 30H and 31H to 01, 02, 0C, 0D, and
;stores them in locations starting from internal RAM location 40H.
REPT:   MOV A, @R0   ;
        ANL A, #0F0H;

        SWAP A       ;
        MOV @R1, A   ;

        MOV A, @R0   ;
        ANL A, #0FH  ;

        INC R1       ;
        MOV @R1, A   ;

        INC R0       ;
        INC R1       ;
        DJNZ R2,REPT;

        MOV R0, #40H;
        MOV R1, #50H;
        MOV R2, #04H; R2 is used as loop counter
```

```
;The 'AGAIN' loop converts the hex number pointed by R0 to its
;equivalent ASCII, and stores in location pointed by R1. This
;is done by adding 30H to hex number, if the number is in the
;range 0 to 9. If the number is in the range A to F, 37H is
;added. The loop is repeated 4 times.

AGAIN: MOV A,@R0
       ADD A, #30H
       CJNE A, #3AH,FRWD

FRWD:  JC NUMB
       ADD A, #07H
NUMB:  MOV @R1, A
       INC R0
       INC R1

       DJNZ R2, AGAIN
EXIT: SJMP EXIT
```

In the previous program it should be noted that the following two instructions of 8051

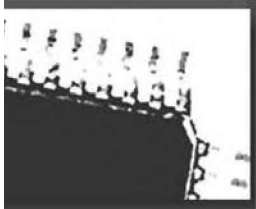
```
CJNE A, #3AH,FRWD
FRWD:  JC NUMB
```

are equivalent to the following 8085 instructions:

```
CPI 3AH
JC NUMB
```

1. What is a microcontroller? What are its typical applications?
2. Mention the salient features of the 8051 microcontroller.
3. Explain the functional pin diagram of 8051 with a neat diagram.
4. Describe program memory structure. With a neat diagram explain interfacing of external program memory.
5. Describe data memory structure.
6. With a neat diagram explain interfacing of 2K bytes of external data memory.
7. Describe internal RAM organization in 8051.
8. Explain the various bits available in PSW register.
9. What happens to various register contents in 8051 after reset?
10. Describe the programmer's view of 8051.
11. Explain with examples the various addressing modes of 8051.
12. Mention the various types of instructions available for 8051, with examples.
13. Explain the working of the following instructions of 8051.

a. XCH A, 40H	b. XCHD A, @R1	c. MOVX A, @DPTR	d. SWAP A
e. MUL AB	f. DIV AB	g. MOVC A,@A+PC	h. CLR C
i. DJNZ R3, BACK	j. CJNE A, 40H, LOC	k. ANL C,/50H	l. JBC 50H, LOOP



30 Advanced Topics In 8051

- Interrupt structure of 8051
  - *IE (interrupt enable) register*
  - *IP (interrupt priority) register*
  - *External interrupts*
  - *Timer interrupts*
  - *Serial port interrupt*
  - *Interrupt handing in 8051*
- Timers of 8051
  - *TMOD register*
  - *Mode 0 operation of timer/counter*
  - *Mode 1 operation of timer/counter*
  - *Mode 2 operation of timer/counter*
  - *Mode 3 operation of timer/counter*
- Serial interface
  - *Mode 0 of UART*
  - *Use of Mode 0 to expand I/O port capability*
  - *Mode 1 of UART*
  - *Mode 2 of UART*
  - *Mode 3 of UART*
  - *Architecture of 8051*
- Structure and operation of ports
  - *General structure of ports*
  - *Internal structure of P0*
  - *Internal structure of P2*
  - *Internal structure of P1*
  - *Internal structure of P3*
- Power saving modes of 8051
  - *Idle mode*
  - *Power down mode*
- Programming of EPROM in 8751BH
  - *EPROM security*
  - *Program verification*
  - *Erasure of EPROM*

This chapter is devoted to the study of advanced topics in 8051, a popular microcontroller chip of the Intel family. In the previous chapter fundamentals of 8051 were dealt with.

■ 30.1 INTERRUPT STRUCTURE OF 8051

The 8051 can be interrupted from the following five sources.

Two external interrupts INTO* and INT1*;
Two internal timer interrupts T0 and T1;
One internal serial port interrupt.

The interrupt vectors (ISS address) for these interrupt sources are shown as follows.

INT0*	0003H
Timer0	000BH
INT1*	0013H
Timer1	001BH
Serial port	0023H

30.1.1 IE (INTERRUPT ENABLE) REGISTER

Each of the interrupt sources can be individually enabled or disabled by programming the IE register. The IE register is an SFR with the address A8H, and is bit addressable, which is shown in Fig. 30.1. It is programmed by the user. The EA (enable all) bit is actually a global disable bit as described below.

EA: 1 = LS 5 bits decide enable/disable of the five interrupts
0 = Disable all five interrupts, irrespective of LS 5 bits

AF			AC	AB	AA	A9	A8	← Bit address
EA	X	X	ES	ET1	EX1	ET0	EX0	

Fig. 30.1
Bit details of
IE register

Thus 'CLR EA' (or equivalently 'CLR IE.7' or 'CLR AF') is functionally the same as the DI (disable interrupts) instruction of 8085. If EA = 0, the LS 5 bits are don't-care bits. The meaning of the other bits is as follows when EA = 1.

EX0: 1 = Enable eXternal interrupt0 i.e. INT0*
0 = Disable INT0*

ET0: 1 = Enable Timer0 interrupt
0 = Disable Timer0 interrupt

EX1: 1 = Enable eXternal interrupt 1 i.e. INT1*
0 = Disable INT1*

ET1: 1 = Enable Timer1 interrupt
0 = Disable Timer1 interrupt

ES: 1 = Enable Serial port interrupt
0 = Disable Serial port interrupt

30.1.2 IP (Interrupt Priority) Register

Each of the five interrupts can be in one of two priority levels. They are priority levels 1 and 0. Priority level 1 is the higher priority and the user sets these priority levels by programming the IP register. It is a bit-addressable SFR with address B8H. Only the LS 5 bits are meaningful in this register. Figure 30.2 shows the bit details of IP register.

			BC	BB	BA	B9	B8 ← Bit address
X	X	X	PS	PT1	PX1	PT0	PX0

Fig. 30.2 Bit details of IP register

PX0: 1 = 1 level Priority for eXternal interrupt 0 (INT0*)
0 = 0 level priority for INT0*

PT0: 1 = 1 level Priority for Timer0 interrupt
0 = 0 level priority for Timer0 interrupt

PX1: 1 = 1 level Priority for eXternal interrupt 1 (INT1*)
0 = 0 level priority for INT1*

PT1: 1 = 1 level Priority for Timer1 interrupt
0 = 0 level priority for Timer1 interrupt

PS: 1 = 1 level Priority for Serial port interrupt
0 = 0 level priority for Serial port interrupt

If all the five interrupts are set to the same priority level, say level 0, and if several of the interrupts are simultaneously active, the order in which they will be serviced is: INT0*, Timer0, INT1*, Timer1, serial port.

If we want the order of service to be INT0*, INT1*, Timer0, Timer1, serial port, then we have to load IP with xxx00101. However, it is not possible to have some particular sequences of service, for example, INT0*, INT1*, serial port, Timer1, Timer0.

When an interrupt of priority level 0 is being serviced, it can be interrupted by an interrupt of priority level 1, but it cannot be interrupted by another interrupt of priority level 0. When an interrupt of priority level 1 is being serviced, it cannot be interrupted by an interrupt of priority level 1 or priority level 0.

30.1.3 External Interrupts

The external interrupts INT0* and INT1* can be individually programmed to be edge-triggered or level-triggered. This is done by programming the TCON register. TCON is the abbreviation for

Timer Control. Although named TCON, it also provides control and status information about the external interrupts. The bit details of TCON register are provided in Fig. 30.3.

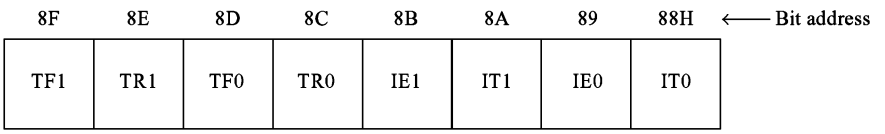


Fig. 30.3 Bit details of TCON register

TCON is a bit-addressable register, with SFR address as 88H. Discussion about the LS 4 bits of TCON that provide control and status information about external interrupts, IT0 and ITI (IT for interrupt type) control bits decide whether INT0* and INT1* will be edge-triggered or level triggered inputs as shown in the following. These bits are programmed by the user.

IT0: 1 = INT0* will be negative edge-triggered input
0 = INT0* will be active low level-triggered input

IT1: 1 = INT1* will be negative edge-triggered input
0 = INT1* will be active low level-triggered input

The IE0 and IE1 bits indicate the status of INT0* and INT1* pins. These bits are set or cleared by 8051. IE stands for 'Interrupt from External source'. For example, IE0 bit is set to 1 by the 8051 when INT0* activation is detected. If INT0* is programmed for edge triggering, IE0 bit is automatically reset to 0 by 8051 when a branch to the ISS at 0003H takes place. If INT0* is programmed for level triggering, IE0 bit is reset to 0 by 8051 only when INT0* input is deactivated. This can be done, for example, by setting to 1 an external flip-flop in the ISS, which deactivates the INT0* request. If this deactivation is not done, the 8051 will again be interrupted after the completion of the ISS.

30.1.4 Timer Interrupts

The MS4 bits of TCON register shown in Fig. 30.3 provide control and status information about the timers as discussed in the following.

TR0 and TR1 (TR for timer run) bits decide the running of these timers. They provide software control over the running of the timers. There can also be hardware control over the running of timers. Thus, even if a timer is set for run mode by software, it will be in run mode only if some hardware conditions are satisfied, as will be discussed later. In other words, TR bit = 1 is a necessary but not sufficient condition for running the timer. TR0 and TR1 bits are programmed by the user.

TR0: 1 = Timer0 is set to run mode
0 = Timer0 is programmed for stop mode

TR1: 1 = Timer1 is set to run mode
0 = Timer1 is programmed for stop mode

TF0 and TF1 bits indicate the overflow status of Timer0 and Timer1. These bits are set or cleared by 8051. For example, TF0 bit is set to 1 by the 8051 when timer0 overflow occurs. TF0 bit is automatically reset to 0 by 8051 when a branch to the ISS at 000BH takes place. More details about the working of timers is provided later.

30.1.5 Serial Port Interrupt

The serial port is used for both transmission and reception. The interrupt status for transmission and reception is provided by TI and RI bits in SCON register. SCON is the abbreviation for serial control. The bit details of SCON register are provided in Fig. 30.4.

9FH	9EH	9DH	9CH	9BH	9AH	99H	98H ← Bit address
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Fig. 30.4 Bit details of SCON register

SCON is a bit-addressable register, with SFR address as 98H. The LS 2 bits of SCON provide status information about serial port interrupts as discussed in the following. TI indicates the transmitter interrupt status, and RI indicates the receiver interrupt status. If any of these 2 bits is a 1, then the 8051 will be interrupted if all the following conditions are met.

(a) ES bit (enable serial port interrupt) = 1 in IE register
(b) EA bit (enable all interrupts) = 1 in IE register
(c) An interrupt of higher or equal priority is currently not being serviced

In such a case, the 8051 branches to the ISS at 0023H. In the ISS, the programmer has to reset the TI and/or RI bit by software. These bits are not automatically reset upon entering the ISS. Further details about the working of serial port and other bits of SCON are provided later. From this discussion on interrupt structure, the 8051 interrupt control system can be visualized as shown in Fig. 30.5.

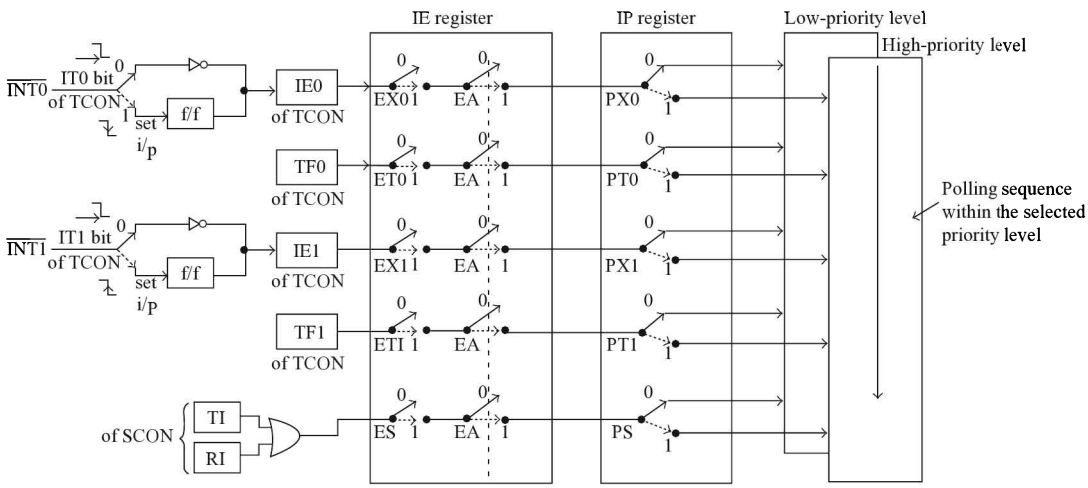


Fig. 30.5 Interrupt control system of 8051

• 30.1.6 Interrupt Handing in 8051

A machine cycle consists of six states, S1 to S6. The interrupt flags IE0, IE1, TF0, TF1 of TCON register and RI and TI of SCON register are sampled at the end of S5 in each machine cycle. This is so,

even if an instruction needs more than one machine cycle for execution. The samples are polled during the next machine cycle. If one of the interrupt flags is set at the end of S5 of the preceding machine cycle, the polling cycle will find it and the interrupt system will generate an LCALL to the appropriate ISS. This happens provided none of the following is true.

(a) An interrupt of equal or higher priority level is already in progress.
(b) The polling cycle is not the final machine cycle in the execution of the instruction in progress.
(c) The instruction in progress is one of the following:
    RETI
    Write to IP register
    Write to IE register

If any of these conditions is true, LCALL to ISS will not be generated. Condition (b) ensures that the instruction in progress is completed before branching to an ISS. Condition (c) ensures that one more instruction is executed after RETI, or any write to IE and/or IP register, before branching to an ISS.

The hardware generated LCALL pushes the contents of the PC on the stack top, and branches to the appropriate ISS. During this ISS, the 8051 can only be interrupted by a higher priority level interrupt. Thus, if the priority level is already 1, it cannot be interrupted till the ISS is completed. The ISS ends with RETI instruction. The RETI instruction pops to PC from the stack top and additionally informs the 8051 that the ISS is over, which causes the interrupt logic to accept additional interrupt at the same priority level as the one just processed. If the ISS is ended with RET instead of RETI, it would have left the interrupt control system thinking that the interrupt was still in progress.

Single-step operation of 8051: The interrupt structure of 8051 allows single-step execution of a program in a simple way as described in the following. Let us say a key is connected to INT0* pin such that when the key is not pressed, the INT0* pin will be in 0 state. Hence on pressing the key, INT0* becomes 1, and on releasing the key later, it becomes 0 again. The key can be designated as the 'single-step' key. Subsequently INT0* interrupt is enabled, INT0* is programmed to be level sensitive, and the ISS for INT0* is terminated with the following code to implement single-step facility. Note that P3.2 bit corresponds to INT0* pin.

```
LOOP1:      JNB P3.2, LOOP1      ; Remain in LOOP1 till INT0* becomes 1.
LOOP2:      JB P3.2, LOOP2       ; Remain in LOOP2 till INT0* becomes 0.
            RETI                 ; Go back and execute 1 instruction.
```

Although the INT0* pin is at 0 for most part of the single-step routine, it does not interrupt 8051 because the same priority interrupt service (in fact, same interrupt) is in progress. Then at the end of the ISS, it waits for pressing and releasing of 'single step' key. After execution of RETI at the end of an ISS, one instruction in the interrupted program will be executed before the 8051 again branches to single step ISS.

■ 30.2 TIMERS OF 8051

There are two 16-bit timer registers in 8051 called Timer0 and Timer1. These timers can be configured to work as times or event counters. In the timer mode, internal machine cycles are counted,

whereas in counter mode external events are counted. Thus the only difference between timer and counter is the source for incrementing the timer register.

In the Timer mode the timer register is incremented for every machine cycle consisting of 12 crystal clock cycles. Thus if the 8051 uses 12-MHz crystal, the timer register is incremented once every microsecond in the timer mode. This mode does not use the external timer input pin. The timer input pin is more appropriately called as counter input pin in this book, although Intel refers to it as timer input pin.

In the counter mode, the timer register is incremented for every 1 to 0 transition at the external counter input pin. Every 1 to 0 transition at the external counter input pin is treated as an event. The external counter input pin is sampled once every machine cycle. Hence, to recognize a 1 to 0 transition, atleast two machine cycles are needed. Thus if the crystal frequency is 12 MHz, the maximum count frequency will be 12 MHz/24 = 500 kHz. In other words, for event counting the time duration between events must be atleast 2 μs.

In addition to selecting as timer or counter, there are four different modes of operation for these timers/counters. Modes 0, 1, and 2 are the same for both the timers/counters, while mode 3 has different meanings for the two timers/counters. The TMOD register is programmed to configure these timers/counters. Whenever serial port is used for serial communication, specifically in mode 1 and mode 3, Timer1 is used for generating the baud rate. In such a case only Timer0 is available for timer/counter operation. Details about serial port are provided later in this chapter.

30.2.1 TMOD Register

TMOD is an SFR with address 89H, and is non-bit addressable. TMOD is abbreviation for Timer Mode. The bit details of TMOD register and the circuit that controls the running of a timer are provided in Fig. 30.6.

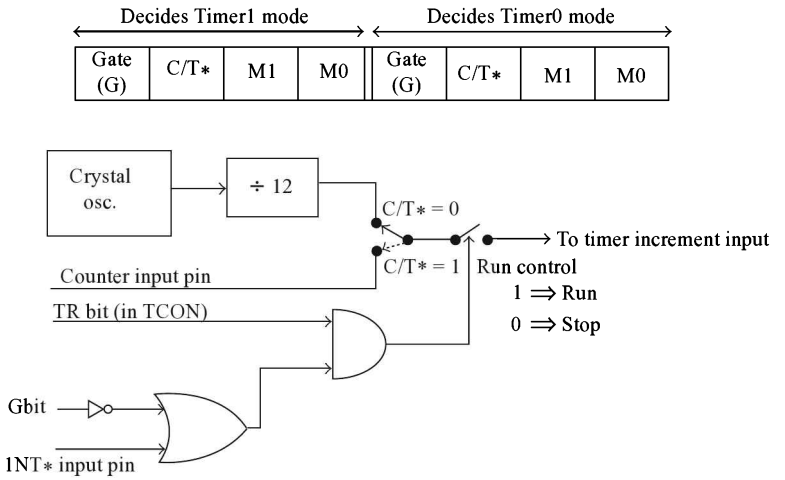


Fig. 30.6 Details of TMOD register and run control of a timer

The user programs all the 8 bits of TMOD, as they are meant for control and not for obtaining status information. The LS 4 bits of TMOD are provided for control of Timer0, and the MS 4 bits of TMOD for control of Timer1, discussed as follows.

C/T*: 1 = Configured for counter operation
0 = Configured for timer operation

M1, M0: 00 = Mode 0 (8-bit timer/counter, with 5-bit prescaler)
01 = Mode 1 (16-bit timer/counter)
10 = Mode 2 (8-bit auto reload timer/counter)
11 = Mode 3 (function depends on if it is for Timer0 or Timer1).

Gate: 0 = TimerX (X = 0 or 1) will be in run mode only if
TRX bit (in TCON register) = 1 (software control)
1 = TimerX will be in run mode only if TRX = 1
and INTX* pin = 1 (hardware control)

This meaning for gate bit is true only when the timer/counter is in mode 0, 1, or 2. The meaning of the gate bit for Timer0 and Timer1 when they are in mode 3 is described later. It is to be noted that in the hardware control of running the timer/counter, external INTX* pin is used. As such, INTX* pin will not be used as an interrupt pin and hence INTX* interrupt should be disabled using IE register.

Example 1
Assume it is desired to use Timer0 as a 16-bit event counter, and Timer1 as an 8-bit auto reload timer. Further, running of these timers is to be controlled by software, then we have to execute the following instruction.

```
MOV TMOD, #25H; Load TMOD with binary 0 0 10 0 1 01
```

Timer1 is configured with gate = 0, C/T* = 0, and M1 M0 = 10 indicating that Timer1 works as a timer in mode 2 (8-bit auto reload mode). Also as gate = 0, the timer starts running when TR1 is set to 1 by software. Timer0 is configured with gate = 0, C/T* = 1, and M1 M0 = 01 indicating that Timer0 works as a counter in mode 1 (16-bit counter mode). Also as gate = 0, the timer starts running when TR0 is set to 1 by software.

Example 2
Suppose it is desired to use Timer0 as a 16-bit event counter, and Timer1 as an 8-bit auto reload timer. Further, running of these timers is to be controlled by hardware, then we have to execute the following instruction.

```
MOV TMOD, #0ADH; Load TMOD with binary 1 0 10 1 1 01
```

Also execute instruction to make TR1 = 1 and TR0 = 1 in TCON register. Now Timer1 is configured with gate = 1, C/T* = 0, and M1 M0 = 10 indicating that Timer1 works as a timer in mode 2 (8-bit auto reload mode). Also as gate = 1, the timer starts running when external INT1* input is at logic 1, providing hardware control.

Timer0 is configured with gate = 1, C/T* = 1, and M1 M0 = 01 indicating that Timer0 works as a counter in mode 1 (16-bit counter mode). Also as gate = 1, the timer starts running when external INT0* input is at logic 1, providing hardware control. In this example, INT1* and INT0* inputs are not used as external interrupt input pins but used to control the running of Timer1 and Timer0 by an external hardware.

30.2.2 Mode 0 Operation of Timer/Counter

Mode 0 operation is an 8-bit timer/counter with a 5-bit prescaler. Figure 30.7 depicts the mode 0 operation as applied to Timer1. Functionally it is a 13-bit timer/counter, using 5 bits of TL1 register and all the 8-bits of TH1 register.

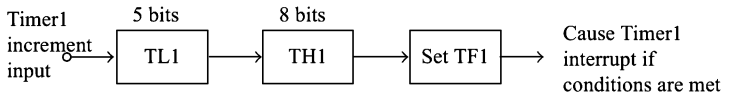


Fig. 30.7
Mode 0 operation
of Timer1

In this case for every $2^5 = 32$ events (in the case of counter operation) or for every 32 machine cycles (in the case of timer operation), the TH1 register is incremented by 1. Finally when TH1 register overflows from FFH to 00, TF1 flag in TCON register is set to 1, and the timer/counter stops. As an example, if TH1 is loaded with F0H, the TF1 flag will be set to 1 after $10H \times 32 = 512$ machine cycles in the timer mode 0. If the crystal frequency is 12 MHz, this happens after 512 μs. The execution of the following instructions generate an interrupt 512 μs after Timer1 starts running.

```
MOV TMOD, #00H  ; Timer1 in Timer mode 0, with Gate = 0
MOV TH1, #0F0H  ; TH1 (MS byte of Timer1) loaded with F0H
MOV IE, #88H    ; Enable Timer1 interrupt
SETB TR1        ; Set TR1 to 1 in TCON register. Start timer.
```

30.2.3 Mode 1 Operation of Timer/Counter

Mode 1 operation is a 16-bit timer/counter. Figure 30.8 depicts the mode 1 operation as applied to Timer0.

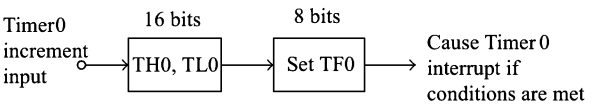


Fig. 30.8
Mode 1 operation of Timer0

In this case, for every event (in the case of counter operation) or for every machine cycle (in the case of timer operation), the TH0–TL0 register pair is incremented by 1. Finally when TH0–TL0 register pair overflows from FFFFH to 0000, TF0 flag in TCON register is set to 1, and the timer/counter stops. As an example, if TH0–TL0 pair is loaded with FFF0H, the TF0 flag will be set to 1 after 10H = 16 machine cycles in the timer mode 1. If the crystal frequency is 12 MHz, this happens after 16 μs. The execution of the following instructions generate an interrupt 16 μs after Timer0 starts running.

```
MOV TMOD, #01H   ; Timer0 in Timer mode 1, with Gate = 0
MOV TL0, #0F0H   ; TL0 (LS byte of Timer0) loaded with F0H
MOV TH0, #0FFH   ; TH0 (MS byte of Timer0) loaded with FFH
MOV IE, #82H     ; Enable Timer0 interrupt
SETB TR0         ; Set TR0 to 1 in TCON register. Start timer.
```

30.2.4 Mode 2 Operation of Timer/Counter

Mode 2 operation is an 8-bit auto reload timer/counter. Figure 30.9 depicts the mode 2 operation as applied to Timer1.

In this case, for every event (in the case of counter operation) or for every machine cycle (in the case of timer operation), the TL1 register content is incremented by 1. Finally when TL1 register overflows from FFH to 00, TF1 flag in TCON register is set to 1. Also TL1 will be reloaded with contents of TH1 and the operation repeats.

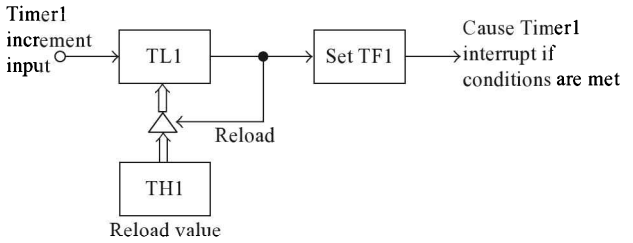


Fig. 30.9
Mode 2 operation
of Timer1

As an example, if TH1 is loaded with F0H and TL1 is also loaded with F0H, the TF1 flag will be set to 1 after 10H = 16 machine cycles in timer mode 2. If the crystal frequency is 12 MHz, this happens after 16 μs. At this point the TL1 is reloaded with F0H, and the operation repeats thus generating an interrupt every 16 μs. The execution of the following instructions generate an interrupt once every 16 μs after Timer1 starts running.

```
MOV TMOD, #20H   ; Timer1 in Timer mode 2, with Gate = 0
MOV TL1, #0F0H   ; TL1 (LS byte of Timer1) loaded with F0H
MOV TH1, #0F0H   ; TH1 (MS byte of Timer1) loaded with F0H
MOV IE, #88H     ; Enable Timer1 interrupt
SETB TR1         ; Set TR1 to 1 in TCON register. Start timer.
```

Timer1 in mode 2 is used for generating the desired baud rate for communication when the serial port is working in mode 1 or mode 3. This aspect will be discussed later.

30.2.5 Mode 3 Operation of Timer/Counter

Mode 3 function is different for Timer0 and Timer1. When Timer0 is operated in mode 3, LS byte of Timer0 (TL0) is used as an 8-bit timer/counter, controlled by the standard Timer0 control bits, Counter0 and INT0* inputs. The MS byte of Timer0 (TH0) is used as an 8-bit timer (not counter) which is controlled by Timer1 control bit TR1. The overflow of TH0 from FFH to 00H causes TF1 (Timer1 overflow) flag to be set. Figure 30.10 depicts the mode 3 operation as applied to Timer0.

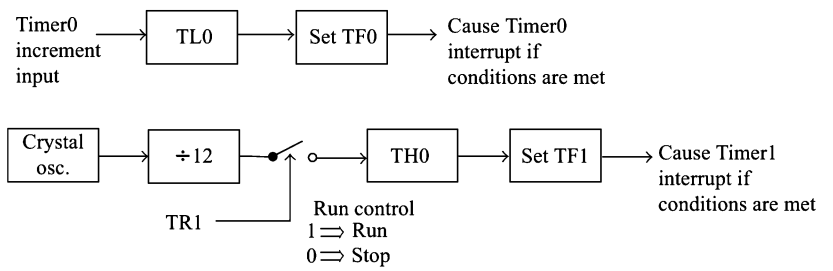


Fig. 30.10
Mode 3 operation of Timer0

When Timer1 is operated in mode 3, it simply holds count and stops running. So when Timer0 is in mode 3, Timer1 is generally configured to work in mode 0, 1, or 2. However, Timer1 will not be in a position to interrupt the 8051, as the TF1 flag (Timer1 overflow) is used by TH0 timer. Timer1 is used as Baud rate generator by the serial port very commonly.

The meaning of the gate bit for Timer0 and Timer1 when they are in mode 3 is as follows. Gate bit for Timer0 in mode 3 controls the running of the 8-bit timer/counter TL0, as in the case of mode 0, 1, or 2. The running of the 8-bit timer TH0 is controlled by TR1 bit only. The gate bit for Timer1 in mode 3 does not have any role, as Timer1 in mode 3 only holds count and stops running.

Mode 3 is provided for applications requiring an extra 8-bit timer/counter. When Timer0 is in mode 3, the 8051 has one 8-bit timer using TH0, one 8-bit timer/counter using TL0, and one 16-bit timer/counter using Timer1. It is to be noted that with Timer0 in mode 3, whenever Timer1 operates in mode 0, 1, and 2 the run control for Timer1 is activated when gate bit = 0 or INT1* external input is a logic 1. The run control is deactivated when gate = 1 and INT1* = 0. It is also deactivated whenever Timer1 mode is set to 3.

■ 30.3 SERIAL INTERFACE

The serial port of 8051 can operate in four modes and the operational mode is decided by SM0 and SM1 bits of SCON register. SCON is the abbreviation for serial control. SCON is an SFR with address 98H, and is bit addressable. The bit details of SCON register, which is provided in Fig. 30.4, is reproduced again in Fig. 30.11.

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Fig. 30.11
Bit details of SCON register

SM0	*SM1*	*Mode*	*Description*	*Baud rate*
0	0	0	Shift register	*f*/12 (*f* = crystal frequency)
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	*f*/64 or I*f*/32
1	1	3	9-bit UART	variable

30.3.1 MODE 0 OF UART

In this mode the serial data bit stream coming into the 8051 on RXD input pin (P3.0 pin) is received. The data is received treating every 8 bit as one unit. A TXD output pin (P3.1 pin) is used to provide the receive clock. The clock frequency sent out by 8051 on this pin is a fixed frequency of *f*/12, where *f* is the crystal frequency. Thus, if the crystal frequency is 12 MHz, a bit is received on RXD pin once every microsecond.

The 8051 has SBUF (serial buffer) register, which is used for serial communication. It is an SFR with address 99H, and is not bit addressable. The SBUF is connected internally to serial port receive register and serial port transmit register, which are not directly accessible by the programmer. The SBUF register is shown in Fig. 30.12. Whenever the 8051 reads the SBUF register, it actually reads from serial port receive register. Similarly, when 8051 writes to SBUF, it will actually be written to serial port transmit buffer.

Reception is initiated when REN (receive enable) bit in SCON = 1, and RI flag bit in SCON = 0. The bits received are accumulated in the serial port receive register, starting with the LS bit. From the point of view of the programmer there is SBUF register, and not serial port receive register. Whenever SBUF is read, it is actually done from serial port receive buffer. Once all the 8 bits are received, the

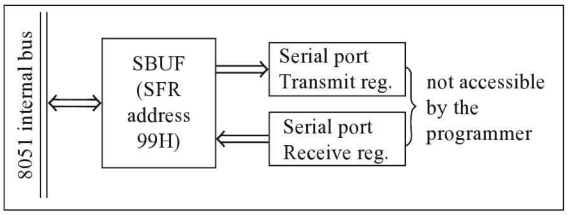


Fig. 30.12
SBUF register in 8051

RI flag is set to 1 by the 8051. The user can perform a status check of this bit to perform the read operation from SBUF. In such a case, the user will disable the serial port interrupt. Alternatively, the RI bit can be allowed to interrupt the 8051 which results in a branch to ISS at 0023H provided that:

(a) EA and ES bits in IE register are set to 1;
(b) Higher or equal priority level interrupts are currently not being serviced.

The RI flag must be cleared by software at the end of the ISS.

Mode 0 can also be used for transmitting the 8-bit data present in serial port transmit register in serial fashion on RXD pin, which now acts as an output pin. The serial port transmit register is written by writing to SBUF. The TXD output pin is used to provide the transmit clock. The clock frequency sent out by 8051 on this pin is a fixed frequency of $f/12$, where f is the crystal frequency. Thus if the crystal frequency is 12 MHz, a bit is transmitted on RXD pin once every microsecond. Transmission is initiated by any instruction which uses SBUF as a destination register, for example, 'MOV SBUF, A'. The bits are transmitted out from the serial port Transmit register, starting with the LS bit. Once all the 8 bits are transmitted the TI flag is set to 1 by the 8051. The user can perform a status check of this bit to perform the next write operation to SBUF. In such a case, the user will disable the serial port interrupt. Alternatively, the TI bit can be allowed to interrupt the 8051 which results in a branch to ISS at 0023H provided that:

(a) EA and ES bits in IE register are set to 1;
(b) Higher or equal priority level interrupts are currently not being serviced.

The TI flag must be cleared by software at the end of the ISS.

As is clear from this discussion, mode 0 is a synchronous mode working in half duplex mode. The most common use of mode 0 is to expand the I/O capability of the 8051 using external shift registers. Mode 0 of UART is also used for high-speed serial data collection. In mode 0 operation of UART, SM2 bit should be 0, as it has no role to play in this mode.

30.3.2 Use of Mode 0 to Expand I/O Port Capability

Serial to parallel shift registers are used for realizing additional output ports. The output mode uses RXD as the serial data output and TXD as the bit-shifting clock for the serial to parallel shift register. Also, an additional output line of 8051 is used to strobe the parallel data out of the shift register. Figure 30.13 depicts the implementation of an external output port.

To implement more output ports, the 'serial out' of the first output port must be connected to the 'serial in' of the second output port and so on. A typical subroutine to output data to the expansion output port using status check data transfer is provided in the following. It is a general program written to output bytes to several expansion output ports. Before branching to this subroutine, load R0 with the number of expansion ports, and RAM buffer with the data to be output. If four expansion ports are

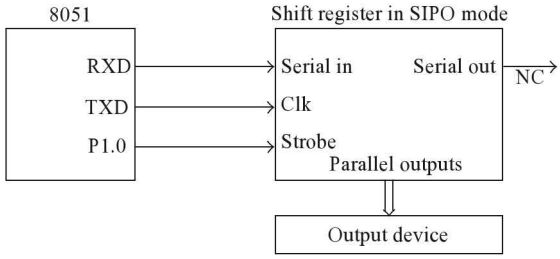


Fig. 30.13
Additional output port
using mode 0 of UART

desired, load R0 with 04. Let us say internal RAM locations 20H to 23H are used as RAM buffer. In such a case, location 23H is loaded with the data to be sent to the first expansion port, and location 20H is loaded with the data to be sent to the last expansion port.

```
        MOV SCON, #00H   ; Select mode 0 operation of UART
        CLR ES           ; Disable serial port interrupt
        MOV R1, #20H     ; Load R1 with starting address of RAM buffer
LOOP:   CLR TI
        MOV SBUF, @R1    ; Start the serial transmission
WAIT:   JNB TI, WAIT     ; Wait in this loop till TI becomes 1
        INC R1           ; Point R1 to next RAM buffer location
        DJNZ R0, LOOP    ; Send all the 4 bytes to expansion ports
        CLR P1.0
        SETB P1.0        ; Generate 0 to 1 transition on strobe line
        RET
```

In this program, a RAM buffer from location 20H is used to store the data to be output to the expansion ports. An advantage of the RAM buffer approach is that if the RAM buffer is chosen to reside in on-chip bit-addressable data RAM, then the user can set, clear or test each bit of an expansion port by simply setting, clearing, or testing the corresponding bit in the RAM buffer.

Similarly, parallel to serial shift registers are used for realizing additional input ports. The input mode uses RXD as the serial data input, and TXD as the bit-shifting clock for the parallel to serial shift register. Also an additional output line of 8051 is used to strobe data from the parallel inputs of the external shift register. Figure 30.14 depicts the implementation of an external input port.

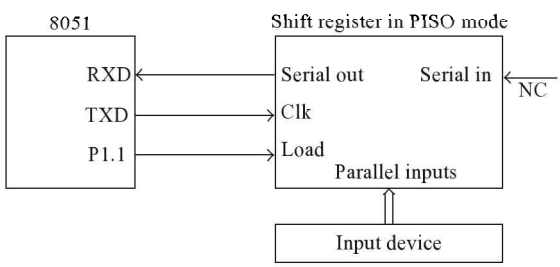


Fig. 30.14
Additional input port
using mode 0 of UART

To implement more input ports, the 'serial in' of the first input port must be connected to the 'serial out' of the second input port and so on. A typical subroutine to input data from the expansion input port is left as an exercise to the reader.

30.3.3 Mode 1 of UART

Mode 1 is used for asynchronous data transfer. It is a full duplex mode, which means that it can transmit and receive simultaneously. RXD input pin is used for receiving the serial data and accumulating it in the SBUF (actually in serial port receive register). It receives the start bit (which is logic 0), then the 8-data bits (bits 0 to 7) starting from the LS bit, and finally the stop bit (which is logic 1). The received start bit is ignored, the data bits 0 to 7 are loaded in SBUF, and the stop bit is loaded into RB8 (bit 8 of received data) of SCON register.

In this mode the serial data bit stream coming into the 8051 on RXD input pin at the programmed baud rate is received. Its value is determined by the Timer1 (never Timer0) overflow rate. Generally mode 2 of Timer1 (auto reload mode) is used for baud rate generation. The Timer1 interrupt should be disabled in this application. The Timer1 can be configured for 'timer' or 'counter' operation, but generally it is configured for timer mode. The baud rate is given by the formula

$$2^{SMOD} \times f/[32 \times 12 \times (256 - TH1)]$$

where TH1 is the content of MS byte of Timer1 register and SMOD is the MS bit in PCON (power control) register. PCON is an SFR with address 87H, and is not bit addressable. Whenever SMOD bit = 1, the baud rate is doubled if the serial port is configured for mode 1, 2, or 3. The SMOD bit has no effect on the baud rate of serial port when it is configured for mode 0. More details about PCON register is provided later. Table 30.1 provides the values of baud rates obtained using the above formula when MS nibble of TMOD = 0010 (Timer1 as 'timer' in mode 2).

Table 30.1 Baud Rates Generated by Timer1 in mode 1 of UART

f (MHz)	*SMOD*	*TH1*	*Baud rate*
12	1	FFH	62,500
11.059	1	FDH	19,200
11.059	0	FDH	9600
11.059	0	FAH	4800
11.059	0	F4H	2400
11.059	0	E8H	1200

Reception is initiated by the incoming start bit if REN bit = 1. Once all the 8-data bits and the stop bit are received, the RI flag is set to 1 by the 8051. In mode 1 operation of UART, if SM2 bit of SCON register is set to 1, then RI bit will not be set to 1, if RB8 = 0. As RB8 holds the stop bit in this mode, it implies that RI bit will not be set to 1 when SM2 = 1 unless a valid stop bit is received by the UART.

The user can perform a status check of the RI bit to perform the read operation from SBUF. In such a case, the user has to disable the serial port interrupt. Alternatively, the RI bit can be allowed to interrupt the 8051, which results in a branch to ISS at 0023H provided that the usual conditions for branching to an ISS are statisfied. The RI flag must be cleared by software at the end of the ISS.

The TXD output pin is used for transmitting the data available in SBUF (actually in serial port transmit register) in serial fashion. It transmits the start bit (which is logic 0), then the 8-data bits (bits 0 to 7) starting from the LS bit, and finally the stop bit (which is logic 1). Transmission is initiated by any instruction that uses SBUF as a destination register.

Once all the 8-data bits and the stop bit are transmitted, the TI flag is set to 1 by the 8051. The user can perform a status check of this bit to perform the next write operation to SBUF. In such a case, the user has to disable the serial port interrupt. Alternatively, the TI bit can be allowed to interrupt the

8051 which results in a branch to ISS at 0023H provided the usual conditions for branching to serial port ISS are satisfied. The TI flag must be cleared by software at the end of the ISS.

Transmission is initiated by any instruction which uses SBUF as a destination register, for example, 'MOV SBUF, A'. In this mode it transmits the serial data on TXD output pin at the programmed baud rate. Its value is determined by the Timer1 (never Timer0) overflow rate, as was described earlier for receive operation.

30.3.4 Mode 2 of UART

Mode 2 is also used for asynchrononus data transfer in full duplex mode. RXD input pin is used for receiving the serial data and accumulating it in the SBUF (actually in serial port receive register). It receives the start bit (which is logic 0), then the 8-data bits (bits 0 to 7) starting from the LS bit, a programmed bit 8, and finally the stop bit (which is logic 1). The received start bit is ignored, the data bits 0 to 7 are loaded in SBUF, the programmed bit 8 is loaded into RB8 (bit 8 of received data) of SCON register and the stop bit is ignored.

In this mode it receives the serial data bit stream coming into the 8051 on RXD input pin at the baud rate of f/32 or f/64, where f is the crystal frequency. When SMOD bit in PCON register is 0, the baud rate will be f/64. When SMOD = 1, the baud rate is doubled to f/32.

Reception is initiated by the incoming start bit if REN bit = 1. Once all the 8-data bits, RB8 bit, and stop bit are received, the RI flag is set to 1 by the 8051. In mode 2 operation of UART, if SM2 bit of SCON register is set to 1, then the RI bit will not be set to 1, if RB8 = 0. This feature is useful in multiprocessor communication, as will be described later.

The user can perform a status check of the RI bit to perform the read operation from SBUF. In such a case, the user has to disable the serial port interrupt. Alternatively, the RI bit can be allowed to interrupt the 8051 which results in a branch to ISS at 0023H provided that the usual conditions for branching to an ISS are satisfied. The RI flag must be cleared by software at the end of the ISS.

The TXD output pin is used for transmitting the data available in SBUF (actually in serial port transmit register) in serial fashion. It transmits the start bit (which is logic 0), the 8-data bits (bits 0 to 7) starting from the LS bit, then TB8 bit of SCON register, and finally the stop bit (which is logic 1). Transmission is initiated by any instruction that uses SBUF as a destination register.

Once all the 8-data bits, TB8 bit and the stop bit are transmitted, the TI flag is set to 1 by the 8051. The user can perform a status check of this bit to carry out the next write operation to SBUF. In such a case, the user has to disable the serial port interrupt. Alternatively, the TI bit can be allowed to interrupt the 8051 which results in a branch to ISS at 0023H provided the usual conditions for branching to serial port ISS are satisfied. The TI flag must be cleared by software at the end of the ISS.

The TB8 bit can be loaded by the programmer with a value of 0 or 1, which will be useful in multiprocessor communication, and will be described later. Alternatively, the P flag bit in PSW register can be moved to TB8 bit, which is useful for transmitting data with even parity. In mode 2, the UART transmits the serial data on TXD output pin at the baud rate of f/32 or f/64, based on SMOD bit value.

30.3.5 Mode 3 of UART

Mode 3 is also used for asynchronous data transfer in full duplex mode. It is the same as mode 2, except the baud rate. The baud rate in mode 3 is variable, like in mode 1.

The RXD input pin is used for receiving the serial data and accumulating in the SBUF (actually in serial port receive register). It receives the start bit (which is logic 0), then the 8-data bits (bits 0 to 7) starting from the LS bit, a programmed bit 8, and finally the stop bit (which is logic 1). The received

start bit is ignored, the data bits 0 to 7 are loaded in SBUF, the programmed bit 8 is loaded into RB8 of SCON register, and the stop bit is ignored.

In this mode it receives the serial data bit stream coming into the 8051 on RXD input pin at the programmed baud rate. Its value is determined by the Timer1 (never Timer0) overflow rate. Generally mode 2 of Timer1 (auto reload mode) is used for baud rate generation. The Timer1 interrupt should be disabled in this application. The Timer1 can be configured for 'timer' or 'counter' operation, but generally it is configured for timer mode. The baud rate is given by the formula

$$2^{SMOD} \times f/[32 \times 12 \times (256 - TH1)]$$

where TH1 is the content of MS byte of Timer1 register and SMOD is the MS bit in PCON register. Reception is initiated by the incoming start bit if REN bit = 1. Once all the 8-data bits, RB8 bit, and stop bit are received, the RI flag is set to 1 by the 8051. In mode 3 operation of UART, if SM2 bit of SCON register is set to 1, then RI bit will not be set to 1, if RB8 = 0. This feature is useful in multiprocessor communication, as will be described later.

The user can perform a status check of the RI bit to perform the read operation from SBUF. In such a case, the user has to disable the serial port interrupt. Alternatively, the RI bit can be allowed to interrupt the 8051 which results in a branch to ISS at 0023H provided that the usual conditions for branching to an ISS are satisfied. The RI flag must be cleared by software at the end of the ISS.

The TXD output pin is used for transmitting the data available in SBUF (actually in serial port transmit register) in serial fashion. It transmits the start bit (which is logic 0), the 8-data bits (bits 0 to 7) starting from the LS bit, then TB8 bit of SCON register, and finally the stop bit (which is logic 1). Transmission is initiated by any instruction that uses SBUF as a destination register.

Once all the 8-data bits, TB8 bit, and the stop bit are transmitted, the TI flag is set to 1 by the 8051. The user can perform a status check of this bit to perform the next write operation to SBUF. In such a case, the user has to disable the serial port interrupt. Alternatively, the TI bit can be allowed to interrupt the 8051 which results in a branch to ISS at 0023H provided the usual conditions for branching to serial port ISS are satisfied. The TI flag must be cleared by software at the end of the ISS.

The TB8 bit can be loaded by the programmer with a value of 0 or 1, which will be useful in multiprocessor communication, and will be described later. Alternatively, the P flag bit in PSW register can be moved to TB8 bit, which is useful for transmitting data with even parity. In mode 3, serial data is transmitted on TXD output pin at the programmed baud rate. Its value is determined by the Timer1 (never Timer0) overflow rate, as was described earlier for receive operation.

Multiprocessor communication: Modes 2 and 3 of UART provide facility for multiprocessor communications. In these modes data bits 0 to 7 are received in SBUF, and one more bit, called RB8, is received in RB8 bit of SCON register. The UART can be programmed so that RI bit of SCON will be set to 1, causing an interupt to the 8051, only if RB8 = 1. This is done by setting to 1 the SM2 bit in SCON.

When the master 8051 wants to transmit a block of data to one of the several 8051 slaves, it first sends out an address byte that identifies the target slave. For an address byte RB8 bit will be a 1. For a data byte the RB8 bit will be a 0. With SM2 = 1 in the SCON register of each of the slave 8051s, no slave 8051 will be interrupted by a data byte. An address byte will interrupt all slave 8051s. Now each of the slaves can examine the received address byte, and check whether the address is meant for that particular slave. The addressed slave will clear its SM2 bit to 0. The other slaves do not clear the SM2 bit to 0. Thus from now on only the selected slave will receive the data bytes sent by the master.

30.3.6 Architecture of 8051

So far we have discussed in detail the various blocks of 8051. Armed with this knowledge, the detailed architecture of 8051 as shown in Fig. 30.15(a) can be understood easily.

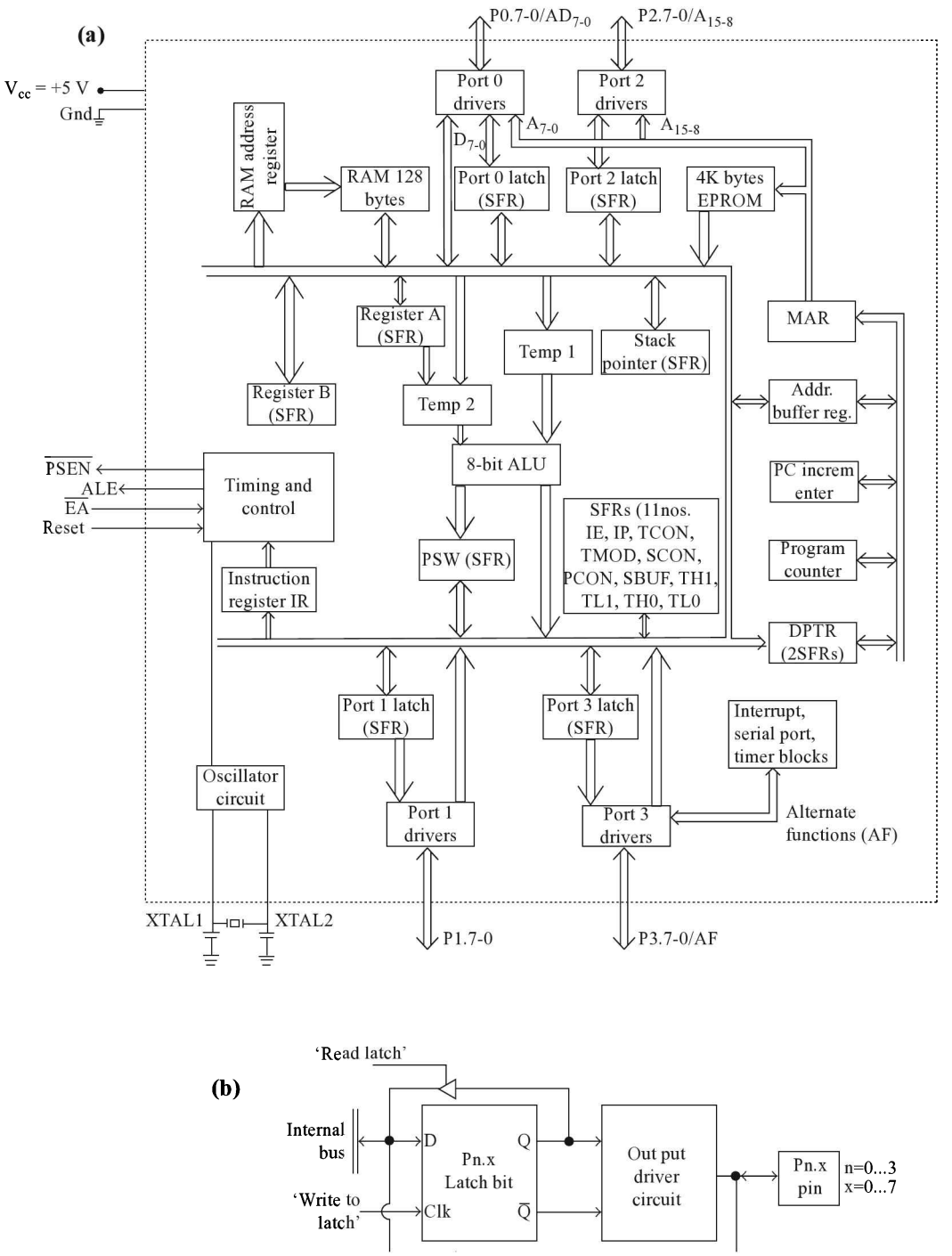


Fig. 30.15 (a) Detailed architecture of 8051; (b) Internal structure common to all ports

■ 30.4 STRUCTURE AND OPERATION OF PORTS

All the four ports—PA0, PA1, PA2, and PA3 are bi-directional. Each port circuit consists of a latch, an output driver, and an input buffer. The latches are bit-addressable SFRs with addresses 80H, 90H, A0H, and B0H for the ports P0, P1, P2, and P3 respectively.

The P1 driver receives the data to be sent out from P1 latch. The P0 and P2 drivers receive the information to be sent out either from the port latch or from the memory address register (MAR). The P3 driver receives information from P3 latch or alternate functions. This can be seen from Fig. 30.15(a), which provides the architecture of 8051.

If external program memory is being accessed to fetch a byte of instruction code, the program counter supplies the address to MAR. The MAR register in turn supplies the address to be sent out to the P0 and P2 drivers. Let us say external program memory is being accessed to fetch a byte from a look up table, using indexed addressing. Then the address buffer register will have the computed value of desired memory address. Hence the address buffer register supplies the address to MAR, which in turn supplies the address to be sent out to the P0 and P2 drivers.

If external data memory is being accessed, using Ri (i = 0 or 1) in register indirect addressing mode, then Ri supplies address buffer register (ABR) with LS byte of address to be sent out. From ABR it is passed on to MAR, which in turn sends it to P0 driver. The P2 latch supplies the MS byte of address to P2 driver. If external data memory is being accessed, using DPTR in register indirect addressing mode, then DPTR supplies MAR with address to be sent out, which in turn sends it to P0 and P2 drivers.

In all the above cases, the P0 pins are used for both sending out LS byte of address and sending/receiving 8-bit data. So these pins are being used as $AD_{7\text{-}0}$ pins in time multiplexed mode. The ALE output from 8051 indicates whether address or data is present on $AD_{7\text{-}0}$ pins. ALE stands for 'address latch enable'. If logic 1 is sent out on ALE, it means 8051 is sending out LS byte of address on P0 pins. An external latch is used to latch this LS byte of address.

In the case of external program memory access, the 8-bit code is received on P0 pins. This information comes to internal bus, and from there is routed to the instruction register (IR), if it is the first or only byte of the instruction. If the data received is the second or third byte of an instruction, they are received in TMP1 and TMP2 registers. IR TMP1, and TMP2 registers are not accessible to the user. In the case of external data memory access, the 8-bit data is received on P0 pins. This information comes to internal bus, and from there is routed to A register.

30.4.1 General Structure of Ports

All the four port structures are very similar. They differ only in the output driver circuit portion. The internal structure common to all the ports is shown in Fig. 30.15b. The output driver circuit portion for the four ports are shown in Figs. 30.16 to 30.19. The internal structure common to all the ports is discussed in this section. The latch bit, which is a bit of the port SFR, is implemented using D-type flip-flop. When the internal 'write to latch' signal is activated, the latch bit is written with the data bit present on D input, which is connected to the internal bus. The output of the flip-flop will be placed on the internal bus in response to internal 'read latch' signal. The level of the port pin itself is placed on the internal bus in response to internal 'read pin' signal.

In some instructions the port latch information is read, and in some others the port pin information is read. When just reading of the port is desired, it is done from the port pins. If it is desired to read, then modify, and finally write, the information will be read from the port latch. Such instructions will

have the destination operand as a port or a port bit. The list of such 'read-modify-write' (RMW) instructions is provided as follows.

```
ANL    (EX. ANL P1, A)
ORL    (EX. ORL P1, A)
XRL    (EX. XRL P1, A)
JBC    (EX. JBC P1.1, LOCN)
CPL    (EX. CPL P1.5)
INC    (EX. INC P1)
DEC    (EX. DEC P1)
DJNZ   (EX. DJNZ P1, BACK)
MOV    (EX. MOV P1.5, C)
CLR    (EX. CLR P1.5)
SETB   (EX. SETB P1.5)
```

For example, in 'ANL P1, A' instruction, the 8051 CPU has to read P1value, then modify it by ANDing it with A contents, and finally write it back to P1. Thus it is a RMW instruction, and so the port latch is read in this instruction. In the 'Mov P1.5, C' instruction, the RMW feature is not obvious. However, the instruction reads all the 8 bits of P1 first, then modifies bit 5 of the port with C value, and then writes the 8 bits back to the port.

The instruction 'MOV A, P1' is not an RMW instruction, as the destination is not a port. So in this case, A register is loaded with P1 pin information. The reason that RMW instructions access the latch rather than the pin is to avoid a possible misinterpretation of the voltage level at the pin. For example, a port bit might be used to drive the base of a transistor. When a 1 is written to the latch bit, the transistor is turned 'on'. Now if the 8051 reads at the pin rather than the latch, it will read the base voltage of the transistor and misinterpret it as a 0. But reading the latch will return the correct value of 1.

30.4.2 Internal Structure of P0

The output driver circuit of a pin of P0 is provided in Fig. 30.16.

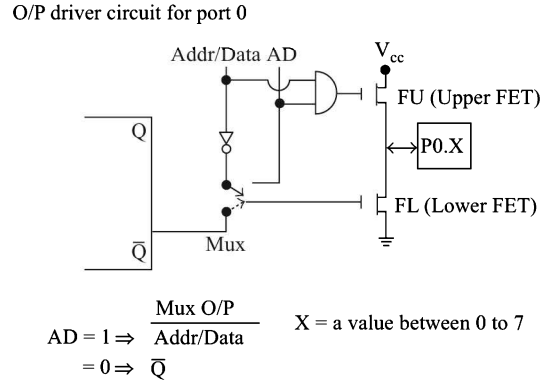


Fig. 30.16
Output driver circuit of a pin of P0

There is a 2 to 1 multiplexer in P0. One of the inputs is from the complement of a bit in the port latch, while the other input is from an Addr/Data bit of MAR after inversion. If the AD internal control signal shown in the figure is a logic 1, the multiplexer outputs the Addr/Data. If P0 is being used as Addr/Data pins, the internal AD signal will be logic 1. If the Addr/Data bit is at logic 1, the upper FET is on, and the lower FET is off. Thus a 1 is sent out as Addr/Data on the port pin. If the Addr/Data

bit is at logic 0, the upper FET off, and the lower FET is on. Thus a 0 is sent out as Addr/Data on the port pin. It is to be noted that there is no need for any external pull up when P0 is being used as Addr/Data pins.

While reading the code byte from external program memory, or reading a data byte from external data memory, the internal AD control signal will be a 0. With the port latch bit at logic 1, both the upper and lower FETs are off. Hence the port pin floats. Now the port pin will be used to receive the code or data byte and the 'read pin' will be activated instead of 'read latch', so that the read will be from the port pin. To facilitate this read operation, the P0 latch will always be written with 1s during access to external program or data memory.

The P0 pins can be used for I/O purposes only if the external program memory or external data memory is not present in the system. In such a case the internal AD control signal will be at logic 0. Then the multiplexer outputs the complement of port latch bit. If the latch bit is at logic 1, the upper FET is off, and the lower FET is also off. Hence the port pin floats. Thus it is seen that P0 does not have internal pull ups when it is used for I/O purposes. So external pull-up resistors have to be used for output operation. Similarly, when the latch bit is at logic 0, the upper FET remains off, and the lower FET is turned on. So the port pin outputs logic 0. If it is desired to use P0 pin as an input pin, the port latch should be written with a 1. Then both the upper and lower FETs are off. Hence the port pin floats, and now it can be used as an input pin.

30.4.3 Internal Structure of P2

The output driver circuit of a pin of P2 is provided in Fig. 30.17.

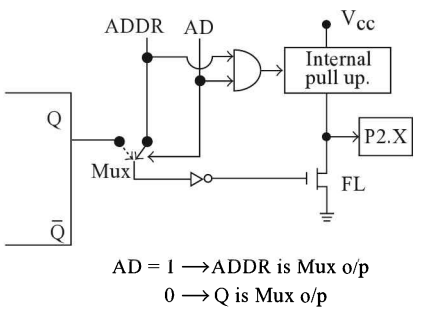


Fig. 30.17
Output driver circuit of a pin of P2

This port uses an active internal pull-up circuit using FETs. The details of this active pull up are not shown for simplicity. There is a 2 to 1 multiplexer in P2. One of the inputs is from a bit of the port latch. The other input is from an Addr bit of MAR. If the AD internal control signal shown in the figure is at logic 1, the multiplexer outputs the Addr. If P2 is being used as Addr pins the internal AD signal will be logic 1. If the Addr bit is a logic 1, the FET is off and the internal active pull up circuit will pull up the port pin to logic 1. Thus a 1 is sent out as Addr on the port pin. If the Addr bit is at logic 0, the internal pull-up circuit will not be active, and the FET is on. Thus a 0 is sent out as address on the port pin. During external memory access, the P2 latch value remains unchanged.

The P2 pins can be used for I/O purposes only if the external program memory or external data memory is not present in the system. In such a case the internal AD control signal will be at logic 0. Then the multiplexer outputs the port latch bit. If the latch bit is at logic 1, the FET is off, and the internal active pull up circuit pulls up the pin output to logic 1. Similarly, when the latch bit is at logic 0, the FET is turned on, and the internal active pull up will not be active. Hence the port pin outputs logic 0. Thus no external pull-up resistors are needed for output operation, whereas for output operation with P0 an external pull-up resistor is needed.

If it is desired to use P2 pin as an input pin, the port latch should be written with a 1. Then the FET is off, and the internal pull-up circuit pulls up the port pin to logic 1. Now the port pin can be used as an input pin. If the external input is at logic 0, the port pin is externally pulled low by this operation.

30.4.4 Internal Structure of P1

The internal structure of a pin of P1 is provided in Fig. 30.18.

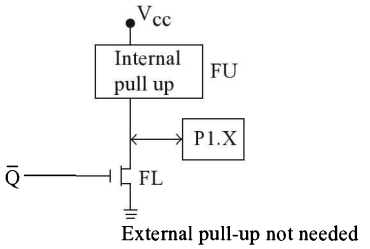


Fig. 30.18
Output driver circuit of a pin of P1

This port also uses an active internal pull-up circuit using FETs. The details of this active pull up are not shown for simplicity. This port is always available for I/O operations immaterial of the existence of external program memory and data memories.

If the latch bit is at logic 1, the lower FET is 'off', and the internal active pulls up circuit pulls up the pin output to logic 1. Similarly, when the latch bit is at logic 0, the lower FET is turned 'on', and the internal active pull up will not be active. So the port pin outputs logic 0. Thus no external pull-up resistors are needed for output operation, whereas for those with port 0, external pull-up resistors are needed.

If it is desired to use P1 pin as an input pin, the port latch should be written with a 1. Then the lower FET is 'off', and the internal pull-up circuit pulls up the port pin to logic 1. Now the port pin can be used as an input pin. If the external input is at logic 0, the port pin is externally pulled low by this operation.

30.4.5 Internal Structure of P3

The output driver circuit of a pin of P3 is provided in Fig. 30.19.

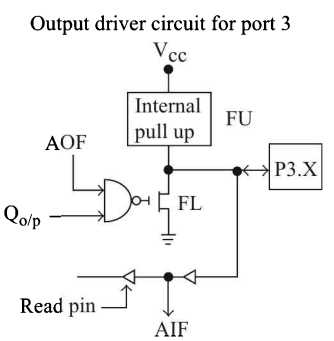


Fig. 30.19
Output driver circuit of a pin of P3

This port also uses an active internal pull-up circuit using FETs. The details of this active pull up are not shown for simplicity. A pin of P3 can be used for I/O operation, if it is not needed for an alternate function. The alternate functions of the P3 pins are listed as follows.

P3.7 RD* (RD* output for reading from external data memory);
P3.6 WR* (WR* output for writing to external data memory);
P3.5 C1 (counter1 external input);
P3.4 C0 (counter0 external input);
P3.3 INT1* (external interrupt 1 input);
P3.2 INT0* (external interrupt 0 input);
P3.1 TXD (transmit data output from serial port);
P3.0 RXD (receive data input for serial port).

In P3, pins 7 and 6 are used as the alternate output function (AOF) signals RD* and WR* for external data memory, whenever external data memory access is performed. For a P3 pin to be used for generating AOF signal, the corresponding port latch bit must be set to 1. Then the AOF internal signal comes out on the port pin. In case the latch bit is at logic 0, the port pin will be stuck at logic 0, which can be easily seen from the figure for P3 structure.

Assume we want a P3 pin to be used as an alternate input function (AIF) signal. For example, we may want pin 2 of P3 to be used as external interrupt 0 input. Then the corresponding port latch bit must be set to 1. Also the internal AOF signal must be at logic 1. Then the lower FET is 'off', and the internal pull-up circuit pulls up the port pin to logic 1. Now the port pin can be used as an AIF pin. If the external input is a logic 0, the port pin is externally pulled low by this operation. The 8051 senses the value of this external AIF signal by sensing the internal AIF signal, as shown in the figure for P3 structure.

If a P3 pin is not needed for alternate function, the port pin can be used for I/O purposes. In such a case the internal AOF signal will be at logic 1. If the latch bit is at logic 1, the lower FET is 'off', and the internal active pull-up circuit pulls up the pin output to logic 1. Similarly, when the latch bit is at logic 0, the lower FET is turned 'on', and the internal active pull up will not be active. Hence the port pin outputs logic 0. Thus no external pull-up resistors are needed for output operation.

If it is desired to use P3 pin as an input pin, the port latch should be written with a 1. The internal AOF signal will be at logic 1. Then the lower FET is 'off', and the internal pull-up circuit pulls up the port pin to logic 1. Now the port pin can be used as an input pin. If the external input is at logic 0, the port pin is externally pulled low by this operation.

■ 30.5 POWER SAVING MODES OF 8051

8051 is typically implemented using HMOS technology, and is available as 8051H. However, the CHMOS version is also available as 80C51. The CHMOS version is used in applications where power consumption is critical, like data collection in a remote place. The CHMOS version provides two power-reducing modes called the idle mode and the power down mode. The typical current drain from the power supply for 80C51 in the various modes are indicated as follows.

Normal mode 18 mA;
Idle mode 2.5 mA;
Power down mode 3 μA.

30.5.1 Idle Mode

The chip is programmed to work in idle mode by setting IDL bit of PCON register to 1. PCON is abbreviation for power control. PCON is an SFR with address 87H and is not bit addressable. The bit details of PCON register is provided in Fig. 30.20. Note that in HMOS devices, only SMOD bit, which is already discussed, is implemented in the PCON register.

SMOD	X	X	X	GF1	GF0	PD	IDL

Fig. 30.20
Bit details of PCON register

Immediately after the IDL bit is set to 1 by an instruction, the device goes to the idle mode. In the idle mode the clock signal is not provided to the CPU core portion. However, the oscillator continues to run, providing the clock to the interrupts, serial port, and timers. This can be seen from Fig. 30.21. Thus the power consumption is greatly reduced, although supply voltage has to be maintained at +5 V. Typically the current drain from the power supply will be only 2.5 mA in idle mode as against 18 mA in normal working mode.

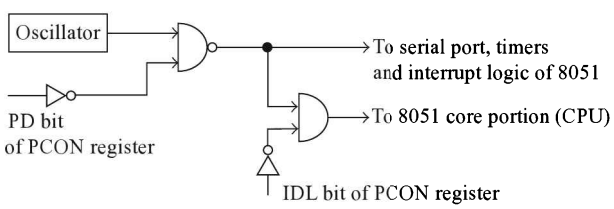


Fig. 30.21
Circuit details of
power saving modes

In this mode the program counter, SP, PSW, accumulator, and all other register maintain their data. In other words, the CPU status is preserved fully. The port pins hold the logic values they had at the time the 8051 went to idle mode. The ALE and PSEN* will be held at logic 1. There are two ways to come out of the idle mode. If a valid interrupt occurs, the IDL bit is reset to 0 by the 8051 automatically. Hence the 8051 comes out of the idle mode. It first services the interrupt and then proceeds with the next instruction after the one that put the 8051 into idle mode.

The action to be taken in the ISS may depend on whether the interrupt occurred during normal operation, or in idle mode. This can be taken care of by the two general-purpose flag bits GF1 and GF0 in PCON register. They can be used as truly general-purpose flags, if desired. However, one of these bits is commonly used to indicate if an interrupt occurred during normal operation, or in idle mode. For example, the instruction that sets IDL bit to 1, can also set GF0 to 1.When the idle mode is terminated by an interrupt, the ISS can check the GF0 bit to take appropriate action. The second method of coming out of idle mode is with a hardware reset. The IDL bit is cleared to 0 when reset. The reset redefines all the SFRs, but does not change the on-chip RAM values.

30.5.2 Power Down Mode

The chip is programmed to work in power down mode by setting PD bit of PCON register to 1. Immediately after PD bit is set to 1 by an instruction, the device goes to the power down mode. In the power

down mode the clock is not provided to any portion of the 8051. Thus all functions within the 8051 are halted. This can be seen from Fig. 30.21. Hence the power consumption becomes extremely low. But the on-chip RAM and SFR values are held. The port pins output the values held by their respective SFRs. The ALE and PSEN* will be at logic 0. Typically the current drain from the power supply will be only 3 μA in power down mode as against 2.5 mA in idle mode.

There is only one way to come out of the power down mode. It is by hardware reset of 8051. The PD bit is cleared to 0 when reset. The reset redefines all the SFRs, but does not change the on-chip RAM values. In the power down mode, the power supply can be reduced to as low as 2 V, but care must be taken to see that power supply is not reduced before going to power down mode. Also power supply should be restored to 5 V before coming out of power down mode.

■ 30.6 PROGRAMMING OF EPROM IN 8751BH

As noted in the previous chapter, 8751 is the EPROM version of 8051. To program the EPROM in 8751, the chip must be running with a 4–6-MHz oscillator. This is because, the internal bus will be used to transfer address and program data to appropriate internal registers. The details of programming of the EPROM vary depending on the 8751 version. They differ with regard to programming voltage value, the width of program pulse, EPROM security features and so on. As such, only programming of the EPROM in 8751BH is described here as a typical example. For other versions of 8751, the reader is advised to go through the manuals of Intel.

The 12-bit address of an EPROM location to be programmed is applied to LS 4 pins of P2 and 8 pins of P1. The data to be written to that location is applied to P0. The connection details for EPROM programming is shown in Fig. 30.22.

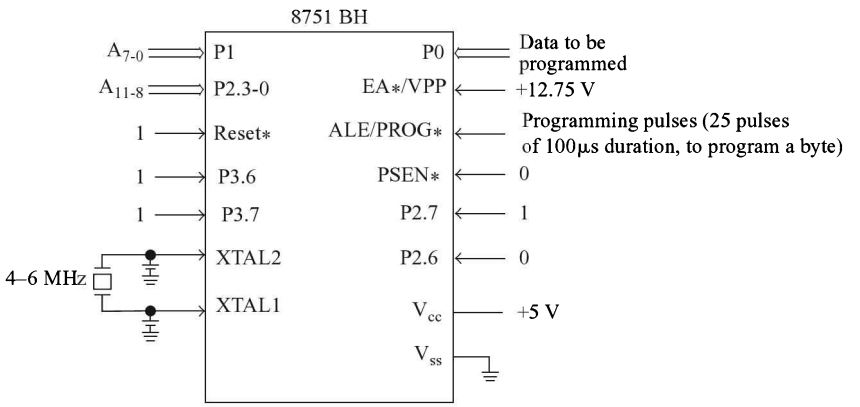


Fig. 30.22 Connection details for EPROM programming

The ALE pin, which is used as an output pin in normal operation, is used as PROG* (program pulse) input during the programming of any part of EPROM, like programming code data, encryption array or lock bits. The encryption array and lock bits are described a little later. The EA* pin, which is tied to 5 V or Gnd during normal operation, is connected to the programming voltage V_{pp} during programming any part of the EPROM. The programming voltage source should be well regulated and free from glitches. Even a narrow glitch above the specified maximum for programming voltage can cause permanent damage to the device.

The 8751BH uses programming voltage of 12.75 V with a margin of 0.25 V on either side. It uses the faster 'quick pulse' programming algorithm. Programming of each byte is done using 25 PROG* pulses, each of 100 μs duration. The gap between each of these 25 PROG* pulses must be atleast 10 μs, as shown in Fig. 30.23. This results in a total programming time of about 13 s for the 4K bytes, taking into account other set up times for programming a byte.

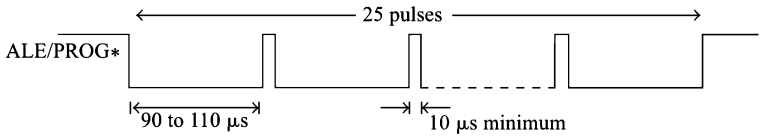


Fig. 30.23
PROG* waveform used in programming a byte

During any of the EPROM operations like program code byte or verify, reset input is tied to logic 1. The PSEN*, which is an output pin in normal operation, works as an input pin and is tied to logic 0 during any of the EPROM operations. The EA*, MS 2 pins of P2, and MS 2 pins of P3 should be held as indicated in 'program code data' row of Table 30.2 for programming a code byte. Pins P2.5 and P2.4 have no role to play in the programming of the EPROM. They are don't-care pins. PROG* is pulsed low 25 times with pulse width of 100 μs to program the code byte into the addressed EPROM location. Normally EA* is held at logic 1, and just before PROG* is to be pulsed EA* is raised to V_{PP}. After PROG* is pulsed 25 times, EA* is returned to logic 1. This completes the programming of 1 byte. The operation is repeated for programming of other bytes.

Table 30.2 Signal Levels for Various EPROM Operations

EPROM operations	*EA**	*P2.7*	*P2.6*	*P3.7*	*P3.6*
Program code data	V_{pp}	1	0	1	1
Program encryption array	V_{pp}	1	0	1	0
Program lock bit 1	V_{pp}	1	1	1	1
Program lock bit 2	V_{pp}	1	1	0	0
Verify code data	1	0	0	1	1
Read signature	1	0	0	0	0

30.6.1 EPROM Security

In many applications it is desirable that the program memory is secured from software piracy. To facilitate such security 8751BH is provided with two program memory lock schemes, which are encryption array and program lock bits.

Encryption array: The EPROM of 8751BH contains a 32-byte array that can be used for the encryption of code data when verification of code data is attempted. The array is initially unprogrammed (all 1s). The user can program the array and for this purpose the address to be sent will be in the range 00 to 1FH to select an encryption byte. The other pins should be held as indicated in 'program encryption array' row of Table 30.2.

The verification procedure sequentially XNORs each byte of the encryption array with a byte of the code. When the last byte in the array is reached, the verification routine restarts with the first byte of the array for the next code byte. Thus the verified code byte will be an encrypted version of the programmed code byte. With the array programmed, the verified bytes can be properly

deciphered to obtain the actual programmed code bytes only if all the 32 bytes of encryption array are correctly known.

If the encryption array is not programmed, the XNOR operation produces a verified code byte that is the same as programmed code byte. Another thing to note is that if the programmed code byte is FFH, the verify operation reveals the encryption byte. Thus, if a large block of EPROM is left unprogrammed, the verify operation will reveal the complete encryption array. Hence it is necessary to program unwanted portion of EPROM also with some random values, to ensure protection of the program. It is recommended that whenever encryption array is programmed, atleast one of the program lock bits is also programmed to ensure stiffer protection. It is to be noted that if both the program lock bits are programmed, the on-chip program memory cannot be read out for the purpose of verification.

Program lock bits: Two program lock bits are provided in the EPROM of 8751BH called LB1 and LB2. If both these bits are unprogrammed, the program lock features are not enabled. However, if encryption array is programmed, verification process provides the program bytes in coded form.

To program the LB1 bit, the signals on the various pins should be held at the levels as indicated in 'program lock bit 1' row of Table 30.2. If only LB1 bit is programmed, MOVC instructions executed from external program memory are disabled from fetching code byte from EPROM of 8751BH. In this mode, EA* is sampled and latched on reset. Thus, if the device is powered up without a reset, the latch initializes to a random value, and holds that value until reset is activated. To ensure proper functionality of the chip, the internally latched value of EA* must tally with its external input. Also, further programming of EPROM in 8751BH is disabled. In this mode, the verification of program code is possible, but unauthorized users cannot easily decipher the program code, as it will be in encrypted form, if the user programs the encryption array. To program the LB2 bit, the signals on the various pins should be held at the levels as indicated in 'program lock bit 2' row of Table 30.2. If both LB1 and LB2 are programmed, the function is the same as in the case where only LB1 is programmed, except that even verification of program code is disabled.

Some versions of 8751, like 87C51, provide three program lock bits—LB1, LB2, and LB3. The function of LB1 and LB2 is the same as in 8751BH. If all the 3 bits are programmed, it denies electrical access by any external means to the on-chip program memory. The device cannot be programmed further and it cannot execute instructions in external program memory. Also, in 87C51, the encryption array is 64 bytes long, double that of 8751BH. Erasing the EPROM deactivates all the lock bits and the device can then be programmed again.

30.6.2 Program Verification

If both the program lock bits are programmed, the on-chip program memory cannot be read out for the purpose of verification. In other cases it is possible to read out program memory contents for the purpose of verification of code. The verification can be done after the programming of every byte, or after the completion of the entire programming operation.

The address of the program memory location is applied to LS 4 pins of P2 and 8 pins of P1. The PROG* pin should be held at logic 1 and the other pins as indicated in 'verify code data' row of Table 30.2. Note that EA* needs to be held at logic 1. The contents of the addressed EPROM location will come out on P0 pins.

Read signature: Read signature function indicates the manufacturer code and the device code for the chip. The method of reading the signature is similar to verifying code data. The PROG* input is held at logic 1, and the other signals are held as shown in the 'read signature' row of Table 30.2. The

address to be sent for reading the signature is 30H and 31H. The signature bytes read will be 89H from location 30H, indicating that the manufacturer is Intel and 51H from location 31H, indicating that the chip is 8751BH.

30.6.3 Erasure of EPROM

The 8751BH comes in a plastic package without a quartz window. As such, its EPROM contents can only be programmed once and cannot be erased. In other words, the 8751BH contains a PROM. However, most other 8751 versions come with a quartz window, and the EPROM contents in these chips can be erased. Exposing the EPROM to an ultraviolet lamp of 12,000 $\mu W/cm^2$ rating for about 30 min, with the EPROM at a distance of about 2 cm is enough to erase the EPROM. This will erase the complete program, encryption array, and the program lock bits. After the erasure, all bits will be in 1 state.

1. Briefly provide an overview of 8051 interrupt structure.
2. Describe the function of the IE and IP registers in 8051.
3. Provide an overview of the timers in 8051.
4. Describe the function of the TCON and TMOD registers.
5. Provide an overview of the serial interface of 8051.
6. Describe with a neat diagram how mode 0 of UART can be used for the expansion of I/O port capability.
7. Provide an overview of the ports in 8051, and their use.
8. With a neat diagram describe the working of P1 in 8051.
9. Describe the power saving modes of 8051.
10. Explain how the EPROM in 8751 is guarded against software piracy.

Bibliography

8080/8085 Assembly Language Programming Manual, Intel Corporation, 1978.

Ayala, K. J. (2005), *The 8051 Microcontroller*, 3rd Ed., Thomson Learning.

Brey, Barry B. (1989), *Microprocessors and peripherals – Hardware, Software, Interfacing and Applications*, 2nd Ed., CBS Publishers & Distributors.

Givone, D. D., and Roesser, R. P. (1980), *Microprocessors/Microcomputers: An introduction*, McGraw Hill.

Leventhal, L. A. (1987), *8080A-8085 Assembly Language Programming*, Osborne/McGraw Hill.

Mathur, A. P. (1995), *Introduction to Microprocessors*, 3rd Ed., Tata McGraw Hill.

Mazidi, M. A., Mazidi, J. G., and McKinlay, R. D. (2006), *The 8051 Microcontroller and Embedded systems*, 2nd Ed., Pearson/Prentice Hall.

Microprocessor Peripheral Handbook, Intel Corporation, 1982.

Motorola 8-bit Microprocessor and Peripheral Data Book, Motorola Inc., 1983.

Peatman, J. B. (1988), *Design with Microcontrollers*, McGraw Hill.

Rafiquzzaman, M. (1990), *Microprocessors and Microcomputer Based System Design*, CRC Press Inc., Reprinted by UBS.

Gaonkar, Ramesh S. (1987), *Microprocessor Architecture Programming and Applications with the 8085/8080A*, Wiley Eastern Limited.

Singh, B. P. (1997), *Microprocessors and Microcontrollers*, Galgotia, 1st Ed.

Short, K. L. (1987), *Microprocessor and Programmed Logic*, PHI.

Shet, K. C., and Hebbar, K. M. (1991), *Microprocessors*, 2nd Ed., CBS Publishers & Distributors.

Zaks, R. (1981), *From Chips to Systems – An Introduction to Microprocessor*, Sybex.

Z80A-CPU technical manual, Zilog Inc., 1978.

This page is intentionally left blank.

4-bit Microprocessors 4
8-bit Microprocessors 4
16-bit Microprocessors 4
32-bit Microprocessors 5
74138 Decoder IC 120
advantage of multiple chip select lines 122
use of 74138 to generate chip select logic 121
8051 Microcontroller 546
8255 Programmable Peripheral Interface Chip 323
architecture of 8255 325
interface with Microprocessor 326
interface with I/O Devices 327
pin diagram of 8255 324
port selection of 8255 325

A

Addressing of I/O ports 125
comparison of I/O port chips and memory chips 126
comparison of memory-mapped I/O and I/O-mapped I/O 129
IN and OUT instructions 127
I/O-mapped I/O 129
memory-mapped I/O 128
need for I/O Ports 125
Addressing modes of 6800 533
Direct addressing 534
Extended addressing 535
Immediate addressing 534
Implied or inherent addressing 534
Indexed addressing 535
Page 0 addressing 534
Relative addressing 536
Addressing modes of 8051 557
advantage of SFR 559
direct addressing 558
immediate addressing 558
implied addressing 560
indexed addressing 559
register addressing 558
register indirect addressing 559
Addressing modes of 8085 59
absolute addressing mode 61
immediate addressing mode 61
implied addressing mode 61
instruction type LDAX rp 62
instruction type STAX rp 62
instruction type LHLD a16 63
instruction type SHLD a16 63
need for addressing mode 60
Register addressing mode 61
Register codes 59
Register indirect addressing mode 61
Addressing modes of Z-80 499
bit addressing 505
bit SET instruction 505
bit RES instruction 505
bit instruction 506
flags registers F 502
indexed addressing 504
opcode size for Z-80 instructions 503
overflow flag 502
relative Addressing 500
ALS-SDA-85M Kit 122
Application of 8212 315
Application of 8212 in mode 0 315
as supplier of eight RST instructions 318
bi-directional bus driver 315
Intel 8212 as gated buffer 315
interrupting input port and Rst *n* interrupt instruction port 316

Application of 8212 in mode 1 321
Intel 8212 as low-order address latch 321
Intel 8212 as an interrupting output port 321
Arithmetic Group of Instructions 65
Architecture of 8259 439
Architecture of 8085 133
address/data buffers 138
Address latch enable signal (ALE) 136
Arithmetic Logic Unit (ALU) 134
Connection of register to the internal bus 139
control signal RD*, WR*, and INTA* 136
incrementer/decrementer 139
internal address latch 139
instruction Register (IR) 138
multiplexer/demultiplexer 138
ready input pin 137
Status Signals IO/M*, S1 and S0 136
Temporary (Temp) Register 138
timing and control unit 134
W and Z registers 138
X1,X2, and Clk out pins 134
Asynchronous Reception 481
Asynchronous Transmission 478
number of bits per character 479
number of clocks for transmitting or receiving a bit 480
parity bit 479
start and stop Bits 480

B

Branch group of instructions 99
IR-Instruction register 101
PC-Program counter 101
W and Z registers 101

C

Chip selection 117
RAM chip-pin details and address range 118
Classification of 8085 instructions 53
Conditional Call Instructions 109
CC a16—Call if carry 110
CM a16—Call if Minus 111
CNC a16—Call if not carry 109
CNZ a16—Call if not zero result 110
CP a16—Call if Positive 110
CPE a16—Call if parity Even 110
CPO a16—Call if parity Odd 110
CZ a16—Call if Zero result 110
Conditional jump instructions 104
call instruction 107
difference between call and jump instructions 107
JC a16—Jump if carry 105
JM a16—Jump if Minus 107
JNC a16—Jump if not carry 104
JNZ a16—Jump if not zero Result 105
JPE a16—Jump if Parity Even 106
JP a16—Jump if Positive 106
JPO a16—Jump if Parity Odd 106
JZ a16—Jump if Zero result 106
RET instruction 107
Conditional Return Instructions 111
RC—Return if carry 112
RM—Return if minus 113
RNC—Return if not carry 111
RNZ—Return if not zero result 112
RPO—Return if party odd 112
RPE—Return if party even 113
RP—Return if positive 113
RSTn—Restart Instruction 113
RZ—Return if zero result 112
Control Port of 8255 328
Interrupt-driven Bi-Directional operation 341
Interrupt-driven input operation 333
Interrupt-driven output operation 335
Interrupt-driven and Status check data transfers 332
Mode 1 – Strobed I/O 331
Mode 2 – bi-directional I/O 340
Mode Definition Control Word 329
Port C bit set/reset control word 330
Status check bi-directional operation 342
Status check input operation 338
Status check output operation 339

D

Data memory structure 551
8051 stack 553
internal data memory organization 552
internal RAM organization 552
programmer's view of 8051 556
PSW register 555
SFR Area 554
Data Transfer group of instructions 52, 53
Instruction type LDA a16 57
Instruction type LXI rp, d16 56
Instruction type MOV r1, r2 54

Instruction type MOV r, M 55
Instruction type MOV M, r 56
Instruction type MVI M, d8 57
Instruction type STA a16 58
Instruction type MVI r, d8 54
Instruction type XCHG 58
Number of instructions in 8085 53
Data transfer schemes 278
about 8085 Interrupts 283
action taken by 8085 when INTR pin is activated 288
action taken when 8085 is interrupted due to a vector interrupt 295
basic or simple data transfer 278
EI and DI instructions 285
execution of 'DAD rp' Instruction 296
handshake data transfer 281
Interrupt- driven data transfer 282
Intr and IntA* Pins 288
Rest-IN* and Reset-Out pins 285
Rst5.5 and Rst6.5 Pins 291
Rst7.5 Pin 292
Status check data transfer 279
Trap Interrupt Pin 293
Decimal addition in 8085 75
BCD numbers 75
DAA instruction 75
Description of 8085 Pins 29
A_{15-8} Pins 31
AD_{7-0} Pins 30
ALE Pin 33
IO/M* Pin 33
RD* and WR* Pins 31
V_{CC} and V_{SS} Pins 30
Description of Matrix keyboard interface 383
program to display scancode of key pressed 382

F

Flags register 66
auxiliary carry flag (AC) 68
instruction type ADI d8 68
instruction type ACI d8 70
instruction type ADC R 69
instruction type INR R 69
parity flag (P) 68
sign flag (S) 68
zero flag 68
Functional blocks of Intel 8051 548
Program Memory Structure 550

G

Generation of .HEX file using a Linker 197
command mode 198
data file mode 198
downloading the machine code to the kit 199
prompt mode 197
running the downloaded program on the kit 201
Generation of .OBJ file using a cross-assembler 195
translation in command mode 197
translation in prompt mode 196

I

Instruction cycle 140
Comparison of different machine cycles 152
I/O Write (IOW) machine cycle 150
I/O Read (IOR) machine cycle 151
Memory Read (MR) machine cycle 144
Memory Write (MW) machine cycle 146
Opcode Fetch (OF) machine cycle 142
Instructions to Perform Addition 66
Instruction type ADD R 66
Instructions to perform 'AND' operation 78
Instruction type ANA R 78
Instruction type ANI d_8 79
Instructions to perform compare operation 83
Instruction type CMP R 83
Instruction type CPI d_8 84
Instructions to perform 'Exclusive OR' operation 80
Instruction to complement accumulator 82
Instruction type CMC 82
Instruction type STC 82
Instruction type XRA R 81
Instruction type XRI d_8 81
Instructions to perform 'OR' operation 79
Instruction type ORA R 80
Instruction type ORI d_8 80
Instructions to perform subtraction 70
Instruction type SUB R 70
Instruction type SUI d_8 71
Instruction type DCR R 72
Instruction type SBB R 72
Instruction type SBI d_8 73
Instruction type DCX rp 74

Instruction type DAD rp 74
Instruction type INX rp 73
Instruction to rotate accumulator 85
Instruction type RLC 85
Instruction type RAL 86
Instruction type RRC 86
Instruction type RAR 87
Instruction set of 6800 536
6800 instructions set summary 540
arithmetic group 537
branch group 538
data transfer group 537
interrupts of 6800 540
logical group 538
miscellaneous instructions 539
Instruction set of 8051 560
arithmetic group 562
bit-processing group 564
data transfer group 560
logical group 563
program branch group 566
Intel 8251A–Universal Synchronous AsynchroNous Receiver Transmitter (USART) 477
Intel 8253–Programmable Interval Timer 461
Control port 464
Description of 8253 timer 462
Internal architecture of a counter 465
Mode 0—interrupt on terminal count 467
Mode 1—re-triggerable mono-stable multi 468
Mode 2—Rate generator 469
Mode 3—Square Wave generator 471
Mode 4—Software-triggered strobe 472
Mode 5—Hardware-triggered strobe 473
Programming the 8253 463
Read on the fly 465
Use of 8253 IN ALS-SDA-85 kit 475
Intel 8257–Programmable DMA Controller 442
Concept of Direct Memory Access (DMA) 442
Description of 8257 DMA Controller chip 444
Need for DMA Data Transfer 443
Intel 8259 A–Programmable Interrupt Controller 416
Interrupt mask register 423
Interrupt request register 423
in-service register 423
need for an interrupt controller 417
overview of the working of 8259 419
pins of 8259 421
registers used in 8259 422
slave register 424
Intel 8279 Keyboard and Display Controller 384
clear Command 400
decoded mode of operation 389
display write inhibit/blanking command 401
end interrupt/error mode set command 402
encoded mode of operation 392
keyboard/display mode set command 399
keyboard interface considerations 388
Interfacing matrix of switch sensors 393
interface strabed input port 394
LED connection details on the ALS kit 394
left entry mode of display 398
pins of 8279 387
program clock command 391
N-key rollover mode 391
status register 390
two-key lockout mode 390
read from display RAM Command 396
read from FIFO/sensor RAM Command 393
right entry mode of display 399
write to display RAM Command 395
Interfacing 7-Segment Display 370
display interface using serial transfer 374
implementation of moving display 375
Interfacing a matrix keyboard 380
Interfacing a simple keyboard 377
Interrupts in 8085 277
Interrupt structure in Z-80 519
int* Interrupt 519
interrupt mode 0 (IM 0) 519
interrupt mode 1 (IM 1) 520
interrupt mode 2 (IM 2) 521
NMT* Interrupt 522
Interrupt structure of 8051 575
external interrupts 576
IE (Interrupt Enable) Register 575
IP (Interrupt Priority) Register 576
interrupt handing in 8051 578
serial port interrupt 578
timer interrupts 577

L

Logical group of instructions 77

M

Main features of intel 8051 547
Main memory 9
  Random access memory (RAM) 9
  Sequential access 9
  Random access 10
Memory speed requirement 153
  27128-20 Compatibility check with 8085AH 158
  assessing compatibility of 27128-20 with 8085AH-2 159
  earliest data output time considering t_{ACC} 156
  earliest data output time considering t_{CE} 158
  earliest data output time considering t_{OE} 158
  wait state generation 160
Microcontrollers and Digital Signal Processors 5
  Input devices 8
  Output devices 8
Microprocessor Kit 41
  Functions performed by a microprocessor kit 41
  changing the contents of a memory location 45
  Checking contents of a memory location 45
  entering the data 48
  entering the program 46
  executing the program in single step mode 49
  Major components of a kit 41
  Major components of ALS-SDA-85m kit 41
  Opcode of an Instruction Mnemonic 42
  Sign-on message 45
Motorola M6800 Microprocessor 529
  pin description of 6800 530
  programmer's view of 6800 531
Multiple memory address range 119

O

Operational modes of 8255 327
  bi-directional handshake I/O 328
  Mode 0 327
  Mode 1 327
  Simple I/O or basic I/O 327
  Strobed I/O or handshake I/O 327

P

Power saving modes of 8051 595
  Power down mode 596
Programming examples 524, 542, 568
  additional of multi-byte numbers 524, 542
  BCD to binary conversion 569
  Binary to BCD conversion 569
  bit manipulation program 571
  block movement without overlap 526, 544
  exchange of blocks 525, 543
  instruction set summary 527
  conversion of four-digit Hex to ASCII 572
  hex to ASCII conversion 571
  shift a multi-byte BCD number to the right 569
Programming of EPROM in 8751 BH 597
  EPROM security 598
  Erasure of EPROM 600
  Program verification 599
Programming the 8251 488
  command instruction 490
  identifying the command in the control port 490
  mode instruction 488
  status port of 8251 491
  use of SOD pin of 8085 for serial transfer 492
Programming the 8257 446
  address registers 446
  Auto load 450
  block chaining operation 451
  cycle stealing data transfer 457
  control register (mode set register) of 8257 449
  count registers 448
  description of the pins of 8257 452
  enabling/disabling of DMA channels 449
  extended write 450
  first/last flip-flop 447
  long-burst mode 457
  repeat block operation 450
  rotating priority 449
  short-burst mode 457
  single-byte transfer 456
  state diagram of 8085 457
  status register of 8257 451
  stop on terminal count (TCS bit) 450
  Working of the 8257 DMA Controller 456

Programming the 8259 with slaves 436
buffered mode of 8259 437
fully nested mode 438
Initialization Command Word 3 (ICW3) 436
Initialization Command Word 4 (ICW4) 437
special fully nested mode of 8259 437
use of 8259 in an 8086-based system 439
Programming the 8259 with no slaves 424
Initialization Command Word 1 (ICW1) 425
Initialization Command Word 2 (ICW2) 425
Initialization Command Word 3 (ICW3) 428
Initialization Command Word 4 (ICW4) 429
Operation Command Word 1 (OCW1) 430
Operation Command Word 2 (OCW2) 430
Operation Command Word 3 (OCW3) 434
polled mode 434
read status of IRR or ISR 435
Special Marked Mode (SMM) 435
Program using 8279 402
ADRDISP Routine 406
BLANK Routine 407
DATDISP Routine 406
Display characters on the kit 408
DISPLAY routine 403
Display scan code of key 408
KBD_RD routine 403
Program to Alternately display and blank 412
Program to blank display 407
Program to change the contents of a memory location 409
Program to check for the availability of display 413
Use of ADRDISP Routine in a program 409
Use of BLANK Routine in a program 409
Use of DATDISP Routine in a program 408
XPAND Routine 405
Program using interface modules 344
decimal counter using logic controller 348
Digital to Analog converter interface 359
Dual slope ADC interface 356
Evaluation of Boolean Expression 346
Generation of rectangular wave using DAC interface 361
Generation of Triangular Wave using DAC interface 362
Logic Controller Interface 344
Simulation of 4-bit ALU 349
Simulation of 8 to 1 multiplexer 351
Stepper Motor Interface 363
Successive Approximation ADC interface 353
Programs using interrupts 302
clear monitor routine 309
Find square of a number using look up table 305
GTHEX monitor routine 308
HXDSP monitor routine 310
Program for adding 2 number input from keyboard 308
Program for adding 4 hex digits of a 16-bit number 309
Program for Decimal Down Counter 306
Program for simulation of throwing a die 302
Program for simulating a stopwatch 303
Programmer's View of 8085 34
Accumulator or Register A 35
Registers B,C,D,E,H and L 36

R

Read Only Memory (ROM) 10
Arithmetic Logic Unit (ALU) 11
Assembly Language Program 14
Central Processing Unit (CPU) 11
EPROM 10
EEPROM or EAPROM 11
High-Level Language Program 15
Input and Output Ports 11
Machine Language Program 13
Mask-Programmed ROM 10
Microcontroller 13
Secondary memory 11
PROM 10
Running the program using the PC as a terminal 201
Add contents of *N* word locations 266
Bubble sort in ascending/descending order as per choice 259
Check for '2 out of 5' code 212

Check for palindrome 228
Compute the HCF of two 8-bit numbers 210
Compute the LCM of two 8-bit numbers 230
Continuing with single step after checking registers/memory 203
Convert 16-bit binary to BCD 250
Convert ASCII to binary 214
Convert binary to ASCII 216
Convert BCD to binary 218
Convert binary to BCD 221
Display alternately 00 and FF in the data field 241
Display memory command 203
Do an operation on two numbers based on the value of *X* 252
Do an operation on two BCD numbers based on the value of *X* 255
Examine registers command 203
Find the smallest number 208
Generation of time delay 239
Multiply two 2-digit BCD numbers 270
Multiply two 16-bit binary numbers 272
Multiply two 8-bit numbers (shift and add method) 268
Running the entire program in a single operation 202
Running the program in single-step mode 202
Search for a number using linear search 206
Selection sort in ascending/descending order as per choice 263
Simulate decimal up counter 237
Simulate decimal down counter 240
Simulate a real-time clock 243
Sort numbers using bubble sort 233
Sort numbers using selection sort 235
Subtract multi-byte BCD numbers 248

S

Serial Interface 584
Architecture of 8051 589
Mode 0 of UART 584
Mode 1 of UART 587
Mode 2 of UART 588
Mode 3 of UART 588
Use of Mode 0 to expand I/O Port capability 585
Signed Binary Integers 18
1's complement notation 20
2's complement fraction 24
2's complement notation 21
sign magnitude notation 18
SIM and RIM instructions 297
HLT instruction 302
Interrupt enable status 300
Mask status of interrupts 300
Need for Masking 297
Pending interrupt status 301
Reset RST7.5 flip-flop 298
SIM Instruction 297
Serial input of data and SID pin 301
Serial output of data and SOD pin 299
Simple assembly language programs 165
add *N* numbers of size 8 bits 178
add two multi-byte BCD numbers 171
add two multi-byte numbers 169
block movement with overlap 175
block movement without overlap 174
check the fourth bit of a byte 181
divide a 16-bit number by an 8-bit number 187
exchange 10 bytes 165
monitor routines 180
multiply two numbers of size 8 bits 184
subtract two multi-byte numbers 182
UPDDT routine 189
UPDAD routine 180
Special instruction types 506
CPD instruction 509
CPDR instruction 510
CPI instruction 509
CPIR instruction 509
DAA instruction 506
LDD instruction 508
LDDR instruction 508
LDI instruction 507
LDIR instruction 507
IND instruction 511
INDR instruction 511
INI instruction 510
INIR instruction 511
IN reg, (C) instruction 510
OTDR instruction 512
OTIR instruction 512
OUT (C), reg instruction 511
OUTD instruction 512
OUTI instruction 512
Pins of Z-80 517
RLD and RRD instructions 513
RLD instruction 514

Special instruction types (*contd.*)
Rotate and shift instructions 513
RRD instruction 514
SRA instruction 514
SRL instruction 515
Stack and the stack pointer 90
instruction type DAD sp 96
instruction type DCX sp 96
instruction type INX sp 95
instruction type LXI, sp, d16 94
instruction type NOP 96
instruction type POP rp 92
instruction type PUSH rp 93
instruction type SPHL 95
instruction type XTHL 95
reading from the stack 91
writing to the stack 92
Steps needed to run an Assembly Language Program 191
Assembly Language Program 192
assembler directives 193
comments 195
creation of .ASM file using a text editor 195
DB directive 193
DW directive 193
END directive 194
EQU directive 194
Labels 195
ORG directive 193
Symbolic address 195
Structure and operation of ports 591
General structure of posts 591
Internal structure of P0 592
Internal structure of P1 594
Internal structure of P2 593
Internal structure of P3 594
Synchronous Reception 484
Pin description of 8251 USART 484
Synchronous Transmission 483

T

Timers of 8051 579
Mode 0 operation of Timer/Counter 581
Mode 1 operation of Timer/Counter 582
Mode 2 operation of Timer/Counter 582
Mode 3 operation of Timer/Counter 587
TMOD Register 580

U

Unconditional jump instructions 102
JMP a16-unconditional direct jump 103
PCHL-unconditional indirect jump 103
Unsigned binary integers 17

W

Working of 8212 311
Intel 8212 in mode 0 314
Intel 8212 in mode 1 314
Pin diagram of 8212 311

Z

Zilog Z-80 Microprocessor 495
comparison of Intel 8080 with Intel 8085 496
programmer's view of Z-80 497
special feature of Z-80 498