

Salvador Pinillos Gimenez

8051 Microcontrollers

Fundamental Concepts, Hardware,
Software and Applications in Electronics



Salvador Pinillos Gimenez
FEI University Center
São Paulo, São Paulo, Brazil

ISBN 978-3-319-76438-2 ISBN 978-3-319-76439-9 (eBook)
<https://doi.org/10.1007/978-3-319-76439-9>

Library of Congress Control Number: 2018936796

© Springer International Publishing AG, part of Springer Nature 2019

Preface

After decades of use in a wide range of applications, with high performance, efficiency and robustness, the 8051-core microcontrollers have reached a privileged position in the world market of microcontrollers, and to this day, they have been integrated/marketed by several manufacturers.

Built by Intel in the 1980s, this project incorporated new features from many other manufacturers (Texas Instruments, Atmel, Philips, Analog Devices, Infineon, etc.), making even more fans. Boosted by its success in the 1990s, this project has caused a true revolution in the field of portable programmable electronic equipment projects.

This microcontroller is currently used in implantable medical devices (IMD) and in embedded systems for cars, trains, airplanes and aerospace applications, motivating the author to reformulate the book entitled 8051-Core Microcontrollers.

This book, whose title is “8051-Core Microcontrollers – Fundamental Concepts and Exercises,” describes the basic concepts of the 8051-core microcontroller hardware and software, as well as some examples of applications, which can be used in the free simulators/compilers and low-cost kits.

It is divided into nine chapters and an appendix (detailed instructions set). The first chapter presents the basic and fundamental concepts for the understanding of computer systems based in microprocessors/microcontrollers. The second chapter highlights the 8051-core microcontroller hardware. The set of instructions and addressing modes are described and discussed in the third chapter. Chapter 4, considered the most important, explains in detail how a software designer of computer systems should proceed to implement a software project regarding a simple and systematic way, using the flowchart design tool.

The author thoroughly presents and discusses the definition, how the subroutines work, and how they can be used in the structuring of the assembly programming language for the implementation of computer systems, so that the reader may become a software designer.

The input and output interfaces and how to manipulate the variables of a computer system based in 8051-core microcontrollers are described in Chap. 6. In Chap. 7, the interruptions of the 8051-core microcontroller are presented, and some examples of applications are given and discussed.

The timers/counters and their programming are discussed in Chap. 8. The serial communication interface, its programming and application examples are discussed in Chap. 9.

This book mainly aims to enable readers to develop intelligent equipment designs with the most advanced hardware and software concepts regarding the 8051-core microcontrollers. Besides, it can provide huge opportunities in the job market, since there is currently a shortage of skilled labour in this area.

This book also encourages readers to put their ideas into practice and can give great subsidies for future entrepreneurs of compact programmable smart devices in the most diverse areas of applications (industrial, commercial and residential automation, entertainment, security, embedded electronics for cars, trains, airplanes, spaces, sound, image, medical equipment in general, etc.).

Another objective is to provide the teachers of this discipline with a complete material, containing theory and exercises that can be simulated or recorded in the kits.

This book can be used in the disciplines of Engineering, such as microprocessors, microcontrollers applied for Electronic, Robotics, Cybernetic, Automation and Control, Mechatronics, Electrical, Telecommunications, Computers and Bioengineering.

São Paulo, Brazil

Salvador Pinillos Gimenez

Contents

1	Fundamental Concepts of Computer Systems	1
1.1	Introduction	1
1.2	Numeral Systems (System of Numeration)	1
1.2.1	Decimal Numeral System (Base 10)	2
1.2.2	Binary Numeral System	2
1.2.3	Hexadecimal Numeral System	5
1.2.4	Binary-Coded Decimal (BCD)	6
1.2.5	ASCII Code (American Standard Code for Information Interchange)	6
1.3	Codes Conversion	8
1.3.1	Conversion from Any Base to Decimal	8
1.3.2	Decimal to Binary Conversion	8
1.3.3	Decimal in Hexadecimal Conversion	10
1.3.4	Binary in BCD Conversion	11
1.3.5	BCD in Binary Conversion	11
1.4	Memory	11
1.4.1	Mask-Programmable Read-Only Memory (MPROM)	13
1.4.2	Programmable Read-Only Memory (PROM)	15
1.4.3	Erasable Programmable ROM (EPROM)	17
1.4.4	Electrically Erasable Programmable Read-Only Memory (EEPROM)	18
1.4.5	Flash Memory	19
1.4.6	Static Random-Access Memory (SRAM)	19
1.4.7	Dynamic Random-Access Memory (DRAM)	20
1.5	Internal Oscillator (Clock)	21
1.6	Reset (Initiation) Circuit of a Computer System	22
1.7	The Input Devices and Their Associated Interfaces for Computer Systems	23
1.7.1	Mechanical Switches	23
1.7.2	Matrix Keyboard	24
1.7.3	Sensors and Their Associated Interfaces	27
1.7.4	Analog-to-Digital Converter	28

1.8	The Output Devices and Their Associated Interfaces for Computer Systems	29
1.8.1	The LED and Their Associated Interfaces	29
1.8.2	Bar LEDs and Its Associated Interface	30
1.8.3	7-Segment Display	30
1.8.4	Liquid Crystal Display (LCD)	34
1.8.5	Pulse Width Modulation (PWM)	36
1.8.6	Activation of DC and Alternating Current (AC) Loads	38
1.8.7	The Digital-to-Analog Converter (DAC)	39
1.9	Microcomputer	41
1.9.1	Microprocessor Instructions	41
1.9.2	Software	42
1.9.3	Hardware	42
1.9.4	Firmware	42
1.9.5	Central Processing Unit (CPU) or Microprocessor or Core	42
1.9.6	Memory Unit	44
1.9.7	Input/Output Units (I/O)	45
1.10	Microcontroller	45
1.11	Flowchart	46
1.11.1	Connection Line	46
1.11.2	Ellipse	46
1.11.3	Rectangle	47
1.11.4	Lozenge	47
1.12	Sequential Programming	47
1.13	Programming with Looping Structure	49
1.14	Structured Programming	52
1.15	Programming Languages	54
1.15.1	Low-Level Programming Language	54
1.15.2	High-Level Programming Language	54
1.15.3	Medium-Level Programming Language	54
1.16	Basic Computer Architecture (Von Neumann Architecture)	55
1.16.1	Address Bus	55
1.16.2	Timing and Control Bus	56
1.16.3	Data Bus	56
1.17	How a Microcomputer Works	57
1.18	Solved Problems	58
1.18.1	Give Examples of Computer Systems	58
1.18.2	What Are the Advantages of Using a Computer System?	58
1.19	Proposed Problems	58
	References	59

2	8051 Core Microcontrollers	61
2.1	Introduction	61
2.2	General Features	61
2.3	Internal Architecture	62
2.4	Pin Description of the 8051 Microcontroller	64
2.5	Memory Organization	64
2.5.1	The Program Memory (Internal and External)	64
2.5.2	The Data Memory	72
2.6	Interface with the External Memory	80
2.7	Mapping Memory and I/O ICs Mapped as Memory	82
2.8	Reset (Initialization) Signal	87
2.9	The Clock Signal	88
2.10	The Machine Cycle	88
2.11	Execution of Instructions Step-by-Step	89
2.12	Low Power Operation	90
2.13	The Power Reduction Mode	90
2.13.1	The Idle Mode	90
2.13.2	The Low Power Mode	91
2.14	Solved Exercises	92
2.15	Proposed Exercises	93
	References	94
3	8051 Microcontroller Instruction Set of the 8051 Core	95
3.1	Introduction	95
3.2	The Special Function Register “Program Status Word” (PSW)	95
3.3	Addressing Modes of the 8051 Core Microcontroller Family	102
3.3.1	Addressing by Register	102
3.3.2	Direct Addressing	103
3.3.3	Indirect or Indexed by Register	103
3.3.4	Immediate Addressing	104
3.3.5	Indirect by Base Registers or Indexed by Base Registers	104
3.3.6	Combined or Mixed Addressing	105
3.4	Instructions of the 8051 Core Microcontrollers Family	106
3.4.1	Instructions Related to the Internal RAM	106
3.4.2	Instructions Related to the External RAM	106
3.4.3	Instructions of the Tables Manipulation	107
3.4.4	Instructions for Arithmetic Operations	108
3.4.5	Instructions for Logical Operations	108
3.4.6	Boolean Instructions	110
3.4.7	Operations of the Unconditional Jump	110
3.4.8	Jumper and Call-to-Condition Operations	112

3.5	Solved Exercises	114
3.6	Proposed Exercises	127
	References	128
4	Flowchart and Assembly Programming	131
4.1	Introduction	131
4.2	General Features of the Assembly Language	132
4.3	Methodology to Implement Software Packages in Assembly	134
4.4	Development of Sequential Software in Assembly	136
4.5	Development of Software with Loop in Assembly	140
4.6	Exercises Solved	144
4.7	Proposed Problems	164
	References	165
5	Subroutine and Structuring of the Assembly Programming Language	167
5.1	Introduction	167
5.2	Definition of the Stack and Queue to Be Used in the Volatile Memory	167
5.3	The Special Function Register Entitled Stack Pointer (SP)	168
5.4	Process of Access (Writing and Reading) a Byte Regarding the Stack	168
5.5	The Process of Calling and Returning of a Subroutine	172
5.6	Resolved Exercises	180
5.7	Proposed Problems	187
	References	189
6	Input/Output Ports of 8051 Core Microcontrollers	191
6.1	Introduction	191
6.2	Input and Output Ports	191
6.3	Some Examples of Programming in Assembly Using the Input/Output Ports	196
6.4	Routines to Generate Time (Timing Routines)	204
6.5	Monitoring Software for Mechanical Keys	207
6.6	Examples of Mechanical Key Monitoring Routines	208
6.7	Resolved Exercises	212
6.8	Proposed Exercises	223
	References	224
7	Basic 8051 Core Microcontroller Interruptions	225
7.1	Introduction	225
7.2	Methods of Variable Management in a Microcontrolled System (Scanning and Interruption)	225

7.3	The Basic Interruption Sources of the 8051 Core Microcontroller	
7.3.1	External Interruptions	229
7.3.2	Interruptions of Timers/Counters 0 and 1 of the 8051 Core Microcontroller	230
7.3.3	The Interruption of the Serial Communication Interface of the 8051 Core Microcontroller	232
7.4	Special Function Registers for Handling Interruptions	232
7.5	Resolved Exercises	236
7.6	Proposed Exercises	239
	References	239
8	Timers/Counters of the 8051 Core Microcontroller	241
8.1	Introduction	241
8.2	Features of the Timers/Counters	241
8.3	Operation Modes of the Timers/Counters	242
8.4	Programming of the Timers/Counters	245
8.4.1	Mode 0	247
8.4.2	Mode 1	249
8.4.3	Mode 2	249
8.4.4	Mode 3	250
8.5	Solved Exercises	251
8.5.1	Proposed Exercises	267
	References	269
9	The Serial Communication Interface of the 8051 Core Microcontroller	271
9.1	Introduction	271
9.2	The Serial Communication Interface of 8051 Core Microcontrollers	271
9.3	Programming/Configuring the Serial Communication Interface	274
9.3.1	Mode 0	275
9.3.2	Mode 1	276
9.3.3	Mode 2	277
9.3.4	Mode 3	278
9.4	Beginning and Ending the Transmission and Reception of a Byte Through the Serial Communication Interface	280
9.5	Solved Exercises	281
9.6	Proposed Exercises	290
	References	290
	Appendix A: 8051 Microcontroller Instructions Set	293
	Index	321

Fundamental Concepts of Computer Systems

This chapter describes the main basic concepts of computer systems.

1.1 Introduction

The majority of electronics products (cellular phones, tablets, notebooks, radios, smart televisions, etc.) are currently implemented with microprocessors, memories and basic electronic circuits, named input/output interfaces, which perform the interaction with the humans. These basic electronic circuits compose the computer systems. So, this chapter aims to describe the fundamental concepts to understand these wonderful machines.

1.2 Numeral Systems (System of Numeration)

The law of numerical formation in a specific base (b) is given by "... $d_m \dots d_3d_2d_1d_0 \cdot d_{-1}d_{-2}d_{-3} \dots d_{-n} \dots$," where d_m and d_n represent a specific digit of the considered numerical base, before and after the point, respectively, and " $m + 1$ " and " n " are the quantity of digits that define the number, before and after the point, respectively. In order to know how this number is given in the decimal base, we need to use Eq. (1.1) [1].

$$(\dots d_m \dots d_3d_2d_1d_0 \cdot d_{-1}d_{-2}d_{-3} \dots d_{-n} \dots)_b = \dots \\ + d_m \cdot b^m + \dots + d_3 \cdot b^3 + d_2 \cdot b^2 + d_1 \cdot b^1 + d_0 \cdot b^0 + d_{-1} \cdot b^{-1} \\ + d_{-2} \cdot b^{-2} + d_{-3} \cdot b^{-3} + \dots d_{-n} \cdot b^{-n} \dots \quad (1.1)$$

For instance, if m is equal to 9, the number has 10 digits before the point (integer part of the number), and if n is equal to 5, the number has 5 digits after the point (fractional part of the number). If there are ten digits before the point ($d_9 \dots d_0$), the

digit d_0 is defined as the most significant digit (MSD), and if there are three digits after the point, d_{-3} is defined as the least significant digit (LSD) [1].

To exemplify, the number 12345.678 in the base 10 (decimal representation) can be represented by using Eq. (1.2) [1]:

$$(12345.678)_{10} = 1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 + 6 \cdot 10^{-1} \\ + 7 \cdot 10^{-2} + 8 \cdot 10^{-3} \quad (1.2)$$

In Eq. (1.2), 10^4 , 10^3 , 10^2 , 10^1 , 10^0 , 10^{-1} , 10^{-2} , and 10^{-3} mean in the decimal representation the tens of thousands, thousand, hundred, ten, unit, tenth, hundredth, thousandth, and so on. Therefore, this number is composed of one ten thousand, two thousands, three hundreds, four tens, five units, six tenths, seven hundredths, and eight thousandths [1].

Similarly, the value in the base 10 of the number 1111.01 in the base 2, i.e., $(1111.01)_2$, is given by Eq. (1.3) [1]:

$$(1010.101)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ = (10.625)_{10} \quad (1.3)$$

The numerical bases that are most commonly used in computer systems are the binary, hexadecimal, and BCD (binary-coded decimal) [1].

1.2.1 Decimal Numeral System (Base 10)

It is the numeral system used by humans, which is composed of ten digits, named “decimal digits”: 0, 1, 2, ..., 9 [1].

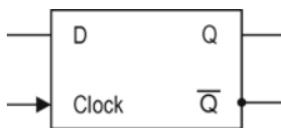
1.2.2 Binary Numeral System

The binary digits or bits are formed by the digits 0 and 1. They belong to the base 2 numeral system. This numeral system is commonly used in computer systems because it is capable of representing the digital electrical behavior of the transistors operating as a switch, i.e., “open switch” (cut-off region: deactivated condition, turn-off, etc.) and “off switch” (triode region: activated condition, turn-on, the transistor is saturated, etc.). Regarding the “logic positive” in digital electronics, the open-switch condition of the transistor is defined as “zero logic” or “0 logic” and the off-switch condition of the transistor is defined as “one logic” or “1 logic.” The 0 logic usually corresponds to the 0 volt (0 V), and the 1 logic corresponds to the 5.0 volts (5.0 V), 3.0 V, 1.2 V, etc., depending on the complementary metal-oxide semiconductor (CMOS) of integrated circuit (IC) technology adopted [1].

The number of binary combinations generated by a bit is equal to $2^{\text{number of bits}} = 2^1 = 2$, i.e., there are two possible binary combinations, which can be 0 logic or 1 logic, as illustrated in Table 1.1 [1].

Table 1.1 Number of binary combinations by using a bit

Binary combinations by using a bit	bit	Decimal value
First binary combination	0	$0 \cdot 2^0 = 0$
Second binary combination	1	$1 \cdot 2^1 = 1$



D input	Clock input (command)	Q output	\bar{Q} output
0	During the raising edge	0	1
1	During the raising edge	1	0
X	Without the raising edge	No change	No change

Nota: X can be 0 or 1 logic.

Fig. 1.1 Electrical representation of the synchronous D-type flip-flop and its truth table

The highest value of the binary combination that can be represented with one single bit is defined as the number of binary combinations subtracted by one, i.e., $2^{(\text{number of bits} - 1)} = 2^1 - 1 = 2 - 1 = 1$, as indicated in Table 1.1 [1].

The synchronous D-type flip-flop (with clock input) is an example of a basic digital circuit which is responsible for storing a single bit, according to Fig. 1.1 [1].

When a bit is defined in the input of a synchronous D-type flip-flop, it will be only copied and stored in the Q output when a rising edge (from 0 to 1 logic) is set in its clock input. The absence of an electrical signal (rising edge) in the clock input does not enable the copy and storage of the bit from the D input to the Q output, and consequently its Q output remains with the previous state. The minimum time of the clock pulse or the maximum frequency that a synchronous D-type flip-flop can operate is determined by the CMOS ICs technology used. The higher the frequency of the flip-flops, the higher the processing speed of computer systems [1].

The numerical representation using 8 bits is defined as a byte. The number of possible binary combinations with 8 bits is 256 ($2^{(\text{number of bits})} = 2^8$), and the highest value of this numerical representation is 255 ($2^{(\text{number of bits})} - 1 = 2^8 - 1 = 256 - 1$), as shown in Table 1.2 [1].

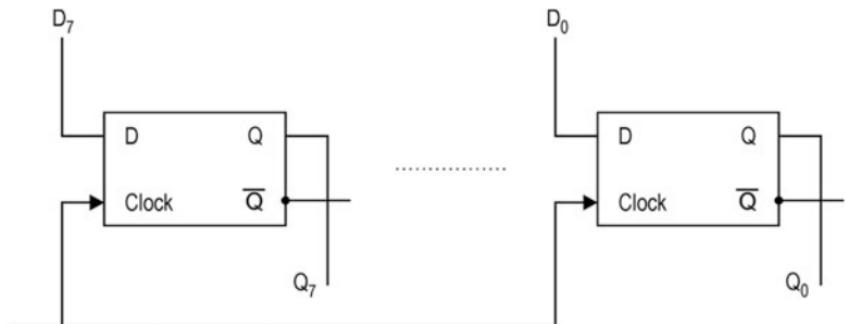
In Table 1.2, D_7, \dots, D_0 are the bits of a byte, in which D_7 is the MSB and D_0 is the LSB of the byte [1].

An 8-bit register is the electrical circuit responsible for storing a byte (8 bits), as illustrated in Fig. 1.2 [1].

In Fig. 1.2, D_7, \dots, D_0 and Q_7, \dots, Q_0 are, respectively, the bits of the inputs and outputs of the synchronous 8-bit register, in which D_7 and Q_7 are the MSB and D_0 and Q_0 are the LSB, respectively [1].

Table 1.2 Possible binary combinations in an 8-bit binary presentation (byte)

Binary combinations in an 8-bit binary presentation (byte)	Byte $D_7D_6D_5D_4D_3D_2D_1D_0$ $2^72^62^52^42^32^22^12^0$	Decimal value
First binary combination	0 0 0 0 0 0 0 0	$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 0$
Second binary combination	0 0 0 0 0 0 0 1	$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1$
Third binary combination	0 0 0 0 0 0 1 0	$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 2$
Fourth binary combination	0 0 0 0 0 0 1 1	$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 3$
Fifth binary combination	0 0 0 0 0 1 0 0	$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 4$
Sixth binary combination	0 0 0 0 0 1 0 1	$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$
:	:	:
Penultimate binary combination	1 1 1 1 1 1 1 0	$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 254$
Last binary combination	1 1 1 1 1 1 1 1	$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 255$

**Fig. 1.2** Example of an electrical representation of a synchronous 8-bit register

To perform a 1-byte write operation on this register, it is necessary to define all the bits of the D inputs (D_7 - D_0); then a single clock pulse is sufficient to perform the write operation on an 8-bit register. As all clock inputs are connected to each other, all inputs D (D_7, \dots, D_0) will be copied to outputs Q (Q_7, \dots, Q_0) at the same time (write operation in parallel) [1].

Usually, we define “word” or “double byte” as a 16-bit set and a “double word” as a 32-bit set. The 16-bit and 32-bit registers are the electrical circuits capable of storing a data of 16 and 32 bits, respectively, similarly to that indicated in Fig. 1.2 [1].

1.2.3 Hexadecimal Numeral System

This numeral system was designed to simplify the write of the data that present a lot of bits. With only two hexadecimal digits, it is possible to represent 1 byte (8 bits). The hexadecimal numeral system consists of 16 hexadecimal digits. Each hexadecimal digit is represented by 4 bits ($0_{16} = 0_h = 0000_2 = 0_{10}$, $1_{16} = 1_h = 0001_2 = 1_{10}$, $2_{16} = 2_h = 0010_2 = 2_{10}$, ..., $9_{16} = 9_h = 1001_2 = 9_{10}$, $A_{16} = A_h = 1010_2 = 10_{10}$, $B_{16} = B_h = 1011_2 = 11_{10}$, $C_{16} = C_h = 1100_2 = 12_{10}$, $D_{16} = D_h = 1101_2 = 13_{10}$, $E_{16} = E_h = 1110_2 = 14_{10}$, $F_{16} = F_h = 1111_2 = 15_{10}$). With only two hexadecimal digits, it is possible to represent 1 byte (8 bits), according to Table 1.3 [1].

Table 1.3 Hexagonal, binary, and decimal representations of a byte

Hexadecimal	Binary	Decimal
00	0000 0000	00
01	0000 0001	01
02	0000 0010	02
03	0000 0011	03
04	0000 0100	04
05	0000 0101	05
06	0000 0110	06
07	0000 0111	07
08	0000 1000	08
09	0000 1001	09
0A	0000 1010	10
0B	0000 1011	11
0C	0000 1100	12
0D	0000 1101	13
0E	0000 1110	14
0F	0000 1111	15
10	0001 0000	16
11	0001 0001	17
:	:	:
19	0001 1001	25
1A	0001 1010	26
:	:	:
1F	0001 1111	31
20	0010 0000	32
:	:	:
9F	1001 1111	159
A0	1010 0000	160
:	:	:
FF	1111 1111	255

Table 1.4 BCD, hexadecimal, decimal, and binary representations of a byte

BCD	Hexadecimal	Decimal	Binary
0000 0000	00	00	0000 0000
0000 0001	01	01	0000 0001
0000 0010	02	02	0000 0010
0000 0011	03	03	0000 0011
0000 0100	04	04	0000 0100
0000 0101	05	05	0000 0101
0000 0110	06	06	0000 0110
0000 0111	07	07	0000 0111
0000 1000	08	08	0000 1000
0000 1001	09	09	0000 1001
0001 0000	10	10	0000 1010
0001 0001	11	11	0000 1011
0001 0010	12	12	0000 1100
0001 0011	13	13	0000 1101
0001 0100	14	14	0000 1110
0001 0101	15	15	0000 1111
0001 0110	16	16	0001 0000
:	:	:	
0001 1001	19	19	0001 0011
0010 0000	20	20	0001 0100
:	:	:	
0010 1001	29	29	0001 1101
0011 0000	30	30	0001 1110
:	:	:	
1001 1001	99	99	0110 0011

1.2.4 Binary-Coded Decimal (BCD)

The BCD digits represent the decimal digits, and therefore they consist of ten digits (from 0 to 9). A BCD digit is composed of a 4-bit binary, as shown in Table 1.4 [1].

1.2.5 ASCII Code (American Standard Code for Information Interchange)

The ASCII, also named ISO 7-bit character code, is extensively used to code alphanumeric characters because the information that a computer handles is not only numeric data, but they can be codes, commands, punctuation marks, graphic signals, and special characters with specific functions to manipulate specific machines, which are called alphanumeric codes. The ASCII table is divided into three parts, ASCII codes which cannot be printed (0–31), ASCII codes (32–127), and extended ASCII codes (128–255), as illustrated in Table 1.5 [1].

DEC	HEX	ASC									
0	0	NUL	64	40	@	128	80	ç	192	C0	+
1	1	SOH	65	41	A	129	81	ü	193	C1	-
2	2	STX	66	42	B	130	82	é	194	C2	-
3	3	ETX	67	43	C	131	83	â	195	C3	+
4	4	EOT	68	44	D	132	84	ä	196	C4	-
5	5	ENQ	69	45	E	133	85	à	197	C5	+
6	6	ACK	70	46	F	134	86	å	198	C6	ä
7	7	BEL	71	47	G	135	87	ç	199	C7	À
8	8	BS	72	48	H	136	88	ê	200	C8	+
9	9	HT	73	49	I	137	89	ë	201	C9	+
10	A	LF	74	4A	J	138	8A	è	202	CA	-
11	B	VT	75	4B	K	139	8B	í	203	CB	-
12	C	FF	76	4C	L	140	8C	î	204	CC	í
13	D	CR	77	4D	M	141	8D	í	205	CD	-
14	E	SO	78	4E	N	142	8E	Ã	206	CE	+
15	F	SI	79	4F	O	143	8F	Å	207	CF	¤
16	10	DLE	80	50	P	144	90	É	208	D0	ð
17	11	DC1	81	51	Q	145	91	æ	209	D1	Ð
18	12	DC2	82	52	R	146	92	Æ	210	D2	Ê
19	13	DC3	83	53	S	147	93	ô	211	D3	Ë
20	14	DC4	84	54	T	148	94	ö	212	D4	È
21	15	NAK	85	55	U	149	95	ò	213	D5	i
22	16	SYN	86	56	V	150	96	û	214	D6	í
23	17	ETB	87	57	W	151	97	ù	215	D7	î
24	18	CAN	88	58	X	152	98	ÿ	216	D8	ï
25	19	EM	89	59	Y	153	99	Ö	217	D9	+
26	1A	SUB	90	5A	Z	154	9A	Ü	218	DA	+
27	1B	ESC	91	5B	l	155	9B	ø	219	DB	-
28	1C	FS	92	5C	\	156	9C	£	220	DC	-
29	1D	GS	93	5D	l	157	9D	Ø	221	DD	í
30	1E	RS	94	5E	^	158	9E	×	222	DE	ì
31	1F	US	95	5F	-	159	9F	f	223	DF	-

DEC	HEX	ASC									
32	20	SP	96	60	'	160	A0	á	224	E0	Ó
33	21	!	97	61	a	161	A1	í	225	E1	ß
34	22	"	98	62	b	162	A2	ó	226	E2	Ô
35	23	#	99	63	c	163	A3	ú	227	E3	Ò
36	24	\$	100	64	d	164	A4	ñ	228	E4	õ
37	25	%	101	65	e	165	A5	Ñ	229	E5	Ӯ
38	26	&	102	66	f	166	A6	ª	230	E6	µ
39	27	'	103	67	g	167	A7	º	231	E7	þ
40	28	(104	68	h	168	A8	¿	232	E8	Þ
41	28)	105	69	i	169	A9	®	233	E9	Ú
42	2A	*	106	6A	j	170	AA	¬	234	EA	Û
43	2B	+	107	6B	k	171	AB	½	235	EB	Ù
44	2C	,	108	6C	l	172	AC	¼	236	EC	ý
45	2D	-	109	6D	m	173	AD	í	237	ED	Ý
46	2E	.	110	6E	n	174	AE	«	238	EE	-
47	2F	/	111	6F	o	175	AF	»	239	EF	'
48	30	0	112	70	p	176	B0	_	240	F0	
48	31	1	113	71	q	177	B1	_	241	F1	±
50	32	2	114	72	r	178	B2	_	242	F2	-
51	33	3	115	73	s	179	B3	í	243	F3	¾
52	34	4	116	74	t	180	B4	í	244	F4	¶
53	35	5	117	75	u	181	B5	Á	245	F5	§
54	36	6	118	76	v	182	B6	Â	246	F6	÷
55	37	7	119	77	w	183	B7	À	247	F7	,
56	38	8	120	78	x	184	B8	®	248	F8	°
57	39	9	121	79	y	185	B9	í	249	F9	"
58	3A	:	122	7A	z	186	BA	í	250	FA	-
59	3B	;	123	7B	{	187	BB	+	251	FB	¹
60	3C	<	124	7C		188	BC	+	252	FC	³
61	3D	=	125	7D	}	189	BD	¢	253	FD	²
62	3E	>	126	7E	~	190	BE	¥	254	FE	-
63	3F	?	127	7F	DEL	191	BF	+	255	FF	DEL

NUL null character, *SOH* beginning of transmission header, *STX* start of text, *ETX* end of text, *EOT* end of transmission, *ENQ* interrogate, *ACK* confirmation, *BEL* beep, *BS* returns a character, *HT* horizontal tabulation, *LF* next line, *VT* vertical tabulation, *FF* next page, *CR* beginning of line, *SO* shift-out, *SI* shift-in, *DLE* data escape link, *D1* device control, *D2* device control, *D3* device control, *D4* device control, *NAK* negative confirmation, *SYN* synchronous idle, *ETB* end of block transmission, *CAN* cancel, *EM* end of the transmission via, *SUB* replace, *ESC* escape, *FS* file separator, *GS* group separator, *RS* register separator, and *US* unit separator [1]

1.3 Codes Conversion

This item describes how we must proceed to convert codes written in different number bases.

1.3.1 Conversion from Any Base to Decimal

To convert a number represented in any base to decimal, just apply Eq. (1.1) [1].

Example 1 Convert 10101100_2 to decimal.

Solution By applying Eq. (1.1), knowing that the base is 2, we have

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
1	0	1	0	1	1	0	0

$$= 10101100_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 172_{10}$$

Example 2 Convert $A5D4_{16}$ ($=A5D4h$) to decimal.

Solution Applying Eq. (1.1), knowing the base is 16, we have

D ₃	D ₂	D ₁	D ₀
16 ³	16 ²	16 ¹	16 ⁰
A	5	D	4

$$\begin{aligned} A5D4_h &= A \cdot 16^3 + 5 \cdot 16^2 + D \cdot 16^1 + 4 \cdot 16^0 = 10 \cdot 16^3 + 5 \cdot 16^2 + 13 \cdot 16^1 + 4 \cdot 16^0 \\ &= 42452_{10} \end{aligned}$$

Example 3 Converter 1234_{bcd} to decimal.

Solution It is the same (1234_{10}) .

1.3.2 Decimal to Binary Conversion

The first method is called sum of powers of 2, in which the digits 0 and 1 are placed in the appropriate positions so that the shares in the sum with their respective weights generate the desired number [1].

Example 1 Convert 100_{10} to an 8-bit binary.

Solution To represent the 8 bits from D_7 to D_0 , define the weight of each digit $2^7 = 128, 2^6 = 64, \dots, 2^0 = 1$ and make the combination of zeros and ones in their respective positions so that the sum gives the value 100_{10} .

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Sum
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
128	64	32	16	8	4	2	1	
0	1	1	0	0	1	0	0	$0.128 + 1.64 + 1.32 + 0.16 + 0.8 + 1.4 + 0.2 + 0.1 = 100_{10}$

The combination of 0s and 1s with their respective weights, $0.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 0.2^0 = 0 + 64 + 32 + 0 + 0 + 4 + 0 + 0$, results in the decimal number 100_{10} [1].

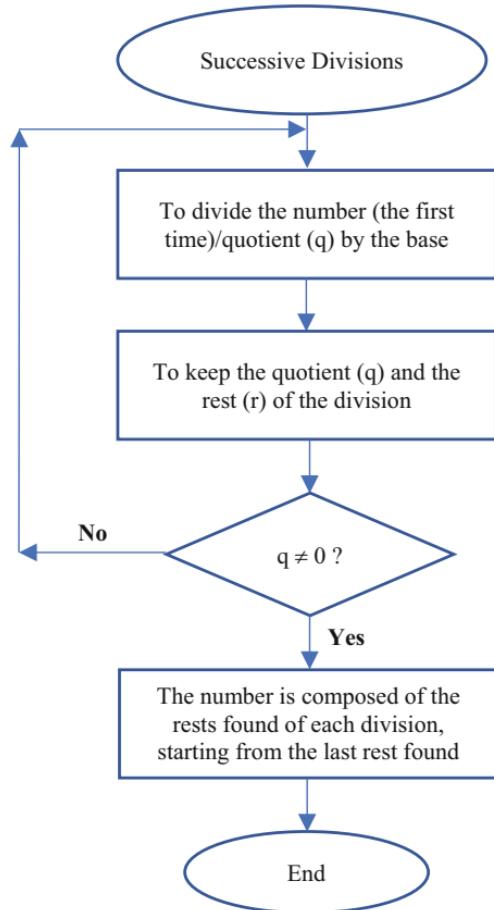
The other method is called “successive divisions,” in which the decimal value must be divided by the number base desired, keeping the rest of each division that will be used to generate the number in the base desired. Besides, the quotient of the previous division must be divided again by the base desired, until the new quotient becomes equal to zero, always keeping the values of the rests of these successive divisions. The converted number will be formed by the rests found, starting from the last rest found. The flowchart of Fig. 1.3 illustrates the methodology of successive divisions [1].

To illustrate, convert the number 100_{10} to 8-bit binary. To implement the solution, divide the number 100_{10} by 2 (desired base). Keep the value of the rest of this division (r_0). Check that the quotient (q_0) is different from zero. Since it is nonzero, divide this quotient (q_0) by 2 again and keep the rest of this new division (r_1). Perform this procedure until the quotient becomes equal to zero. Arrange the rests of the successive divisions as described above [1].

Comments	Division	Quotient (q)	Rest (r)	Bit
	$100/2$	$50 (q_0)$	$0 (r_0)$	2^0 (LSB)
	$50/2$	$25 (q_1)$	$0 (r_1)$	2^1
	$25/2$	$12 (q_2)$	$1 (r_2)$	2^2
	$12/2$	$6 (q_3)$	$0 (r_3)$	2^3
	$6/2$	$3 (q_4)$	$0 (r_4)$	2^4
	$3/2$	$1 (q_5)$	$1 (r_5)$	2^5
End of the successive divisions, due to q_6 being equal to zero	$1/2$	$0 (q_6)$	$1 (r_6)$	2^6 (MSB)

The result is given by a binary number consisting of 7 bits which is equal to 1100100_2 . Since an 8-bit binary was requested, only one zero should be added to the left of that binary number. Therefore, the result is given by $01100100_2 = 64_{16}$ [1].

Fig. 1.3 Flowchart of the successive division methodology



1.3.3 Decimal in Hexadecimal Conversion

The first technique to perform this conversion is the sum of powers, in which the 16 base must be considered, as presented early. The hexadecimal digits must be placed in the appropriate positions so that the sum be equal to the desired number [1].

Example 1 Convert 100_{10} to a 2-digit hexadecimal.

Solution To represent the two hexadecimal digits D_1 and D_0 , the values of weights corresponding to each hexadecimal digit ($D_1: 16^1 = 16$ and $D_1: 16^0 = 1$) must be adjusted in order to obtain the sum equal to 100_{10} .

D_1	D_0	Sum
$16^1 = 16$	$16^0 = 1$	
6	4	$6 \cdot 16^1 + 4 \cdot 16^0 = 6 \cdot 16 + 4 \cdot 1 = 96 + 4 = 100_{10}$

The result is equal to 64_{16} .

Applying the second technique that is illustrated in Fig. 1.3, we have

Comments	Division	q	r	digit
	100/16	6 (q_0)	4 (r_0)	16^0 (LSB)
End of successive divisions	6/16	0 (q_1)	6 (r_1)	16^1 (MSB)

Then the number in hexadecimal that corresponds to the number 100_{10} is equal to $64_{16} = 64_{16}$.

1.3.4 Binary in BCD Conversion

It is a digital codification of the computer systems in order to represent a decimal number, which its decimal digit is represented by a binary number composed of 4 bits. For instance, the binary $2F_h$ or 00101111_2 to be represented in BCD code, we have to transform this number in decimal, which in this case is equal to 47_{10} . After, each decimal digit must be represented in binary, i.e., $0100\ 0111_2 = 47_{BCD}$ [1].

1.3.5 BCD in Binary Conversion

As the BCD value is a decimal value, we have to use the procedure describe in Sect. 1.3.2. To illustrate, consider the number 45_{BCD} . First transform the BCD number in decimal, i.e., 45_{10} . After, we have to transform the decimal value in binary that is equal to 00101101 that is equal to $2D_h$ [1].

1.4 Memory

Its main function is to store digital data (binary numbers), which may be codes of software or data of any type, such as code conversion tables, age of persons, values corresponding to the customer's checking account, predefined temperature values at which a machine must operate, a set of commands that controls the operation of any machine (forward, puncture, stop, inject, etc.), a byte obtained by reading a temperature sensor interface, etc. Memories are of supreme importance because they are an integral part of the basic structure of computer systems [1–11].

There are different types of memories. They can present non-volatile characteristics (they do not lose their data when they are de-energized: read-only memory (ROM), programmable ROM (PROM), erasable PROM (EPROM), electrical EPROM (EEPROM, Flash)) or volatile (they lose their data when they are de-energized: random-access memory (RAM), static RAM (SRAM) composed of D flip-flops, and dynamic RAM (DRAM) composed of capacitors) [1–11].

Memories are made by information storage elements, i.e., keys, flip-flops, capacitors, etc. They are accessed by three different groups of signals, called buses (set of electrical connection paths/wires). The address bus is responsible for setting

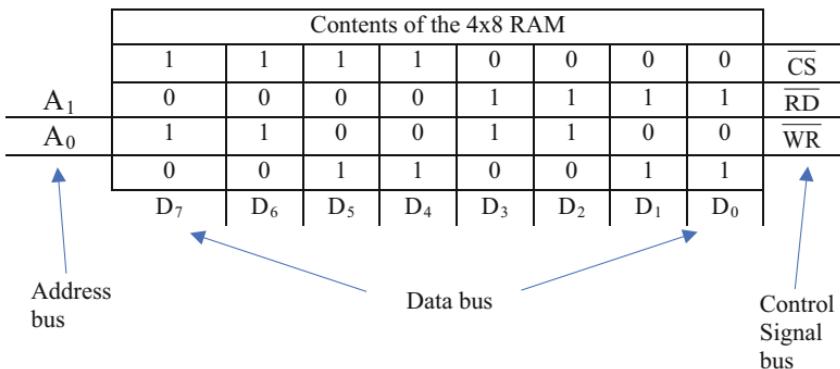


Fig. 1.4 Basic architecture of the 4×8 RAM

the memory location that will be accessed (read or written, depending on the type of memory). The data bus is responsible for making available data to be read from the contents of the memory or to be written in the contents of the memory [1–11].

The control signal bus is responsible for memory general control. An example is the “chip select” (CS) or the “chip enable” (CE). These control signals are responsible for forcing the high-impedance characteristics in the output signals of memories, which are usually named three-state (3-state) conditions. Other examples are the “read” and “write” control signals. They are responsible for defining (triggering) the read and write operations of the memory [1–11].

To clarify, Fig. 1.4 illustrates a 4×8 random-access memory (RAM). The 4 of 4×8 means that they can store 4 contents in the storage cells, e.g., D flip-flop. The 8 of 4×8 means that there are 8 D flip-flops for each content or each set of storage cells has 8 D flip-flops [1, 10].

The quantity of address bits of this memory can be calculated based on the number of storage cells. In this case, this memory can store 4 contents (bytes), thus decomposing the 4 in power of 2, i.e., $4 = 2^A$, where A is the number of bits of address bus, and $2^2 = 2^A$, whose result is $A = 2$. Usually, we call these 2 bits A_1 and A_0 . They are responsible for defining what memory position will be accessed/selected. To access the first memory position, it is necessary to define that A_1A_0 be equal to 00_2 ; to access the second memory content, A_1A_0 must be equal to 01_2 ; A_1A_0 must be equal to 10_2 to access the third location of the memory content; and A_1A_0 must be equal to 11_2 to select the fourth location of the memory content, as shown in Table 1.6 [1, 10].

To illustrate its operation, in order to perform a read operation of the third memory location, first of all it is necessary to define that \overline{CE} be equal to 0 to put the data bus out of the 3-state condition (high impedance). Secondly, we must define that A_1A_0 be equal to 10_2 (address value equal to 2_{10}) to select the third memory content. Afterwards, the command read must be activated, making the signal $\overline{RD}=0$. Regarding these procedures, the contents of the third memory location ($11001100_2 = CCh$) are made available on the data bus (D₇-D₀). We should

Table 1.6 Values of the address bus (A_1A_0) to access the 4×8 RAM contents

Address bus in binary (A_1A_0)	Address value in decimal	Memory location	Content of the memory location
00	0	First	$11110000_2 = F0h$
01	1	Second	$00001111_2 = 0Fh$
10	2	Third	$11001100_2 = CCh$
11	3	Fourth	$00110011_2 = 33h$

highlight that a read operation in a content defined by a memory address preserves its contents, as it only is made available in the data bus and not changed [1, 10].

Besides, to execute a write operation of the BEh (10111110_2) byte in the first memory location, firstly it is necessary to define that the \overline{CE} control signal be equal to 0 in order to put the data bus out of the 3-state condition (high impedance). Subsequently, it is necessary to select the first memory location, making the address bus A_1A_0 equal to 00_2 . Then, we must define the BEh byte on the data bus ($D_7 - D_0$). Finally, we have to define that the \overline{WR} control signal be equal to 0 in order to store/trigger the 10111110_2 (BEh) value in the content of the first memory position. We can observe that the content of the first memory position changes from 11110000_2 (F0h) to 10111110_2 (BEh), i.e., a write operation is able to change the previously stored data in the considered memory position [1, 10].

It is important to highlight that this knowledge described in this section regarding memories are fundamental to understand computer systems.

1.4.1 Mask-Programmable Read-Only Memory (MPROM)

It is a non-volatile ROM, and its programming is done by the manufacturer, which uses a mask of the manufacturing process (usually the metallization mask). The metallization mask is generated by a photolithographic process, which is inherent to the implementation of CMOS ICs. This mask defines the voltage logic levels (0 or 1) of each bit of the content of a specific memory location. These bits and consequently the bytes are performed by the CMOS IC manufacturer as a function of the codes (software/data/information to be recorded) sent by the customer. Due to the fact that it is a dedicated process for each customer, it is expensive to be used in small production volumes. Besides, this type of memory is commonly used in the final production of electronics regarding large volume. This semiconductor device cannot be reprogrammed, characterizing its main disadvantage. Figure 1.5 illustrates the simplified internal architecture of a MPROM 3×8 , implemented with 64 n-channel MOSFETs [1, 10].

In Fig. 1.5, $Q_{07}, \dots, Q_{00}, \dots, Q_{77}, \dots, Q_{70}$ are nMOSFETs, $R_{\text{pull-down}}$ are the pull-down resistances used to avoid the short circuit between V_{dd} and ground, \overline{EN} is the enable pin, $A_2A_1A_0$ is the address bus, D_7, \dots, D_0 is the data bus, and V_{dd} is the supply voltage.

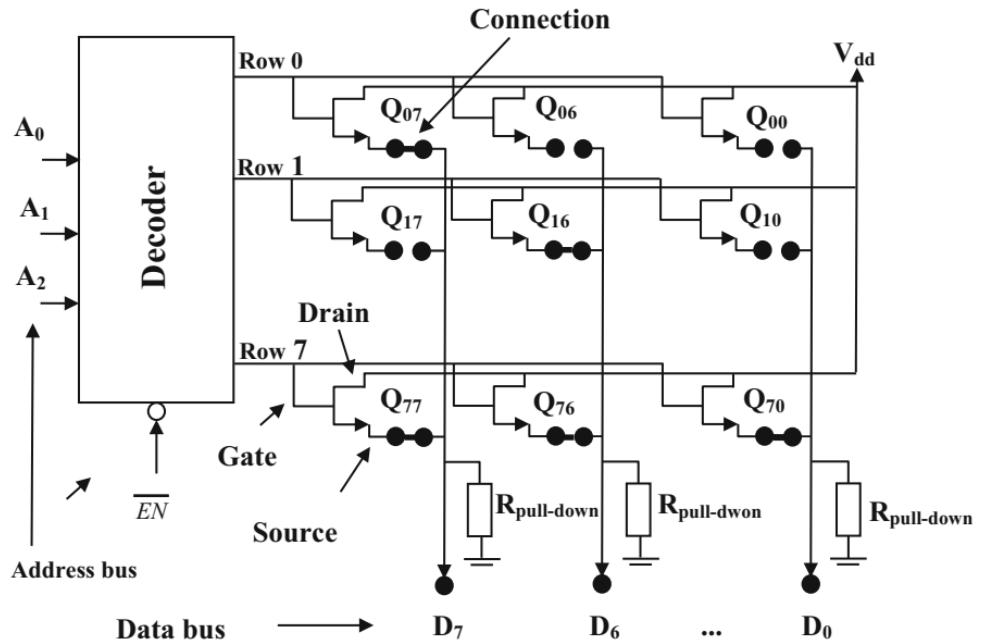


Fig. 1.5 Simplified internal architecture of an 8×8 MPROM

The MROM is composed of three parts: (I) address bus, which is composed of 3 bits (A_2-A_0) and $2^{\text{number of inputs}} = 2^3 = 8$ outputs (Row 7- Row 0). This bus is responsible for activating in logic 1 only one output at a time (address decoder). Each line is connected to the gates of a set of n-channel MOSFETs (nMOSFETs). These transistors are responsible for defining the MROM outputs. Note that the source terminal of Q_{07} is connected to the pull-down resistor ($R_{\text{pull-up}}$) of column D_7 , the source terminal of Q_{06} is connected to the pull-down resistor of column D_6 , and so on. Table 1.7 depicts the truth table of the MPROM with an address bus of 3 bits and 8 outputs, showing the transistors that are activated with their respective MPROM outputs [1, 10].

When all address inputs (A_2-A_0) are set to 000_2 , row 0 is activated and all nMOSFETs' gates ($Q_{07}-Q_{00}$) are set to logic 1, short-circuiting only the drain with the source of Q_{07} , due to the presence of a connection (short circuit) between the source and the pull-down resistor ($R_{\text{pull-down}}$) of the output D_7 , and consequently activating only the output D_7 in logic 1. The other outputs are logic 0 because there are no connections between the source terminals of the transistors and the pull-down resistors ($R_{\text{pull-down}}$), generating an output of 8 bits (D_7-D_0) equal to 10000000_2 . Analogously, when the address inputs A_2-A_0 are set to 001_2 , row 1 is activated at logic 1. Consequently, all gates of $Q_{17}-Q_{10}$ are set to logic 1. Since there is only one connection between the source of the Q_{16} and the $R_{\text{pull-down}}$, the output of the MPROM will be equal to 01000000_2 and so on. Although not shown in Fig. 1.5, by analyzing the MPROM outputs, we suppose that there are also connections between the sources of $Q_{25}, Q_{34}, Q_{43}, Q_{52}$, and Q_{61} and the $R_{\text{pull-down}}$, when the

Table 1.7 Truth table of the MPROM 3×8

Address bus			Row	Activation	Output content (D_7-D_0)
A_2	A_1	A_0			
0	0	0	0	$Q_{07}-Q_{00}$	10000000
0	0	1	1	$Q_{17}-Q_{10}$	01000000
0	1	0	2	$Q_{27}-Q_{20}$	00100000
0	1	1	3	$Q_{37}-Q_{30}$	00010000
1	0	0	4	$Q_{47}-Q_{40}$	00001000
1	0	1	5	$Q_{57}-Q_{50}$	00000100
1	1	0	6	$Q_{67}-Q_{60}$	00000010
1	1	1	7	$Q_{77}-Q_{70}$	11111111

inputs $A_2A_1A_0$ are equal to 010_2 , 011_2 , 100_2 , 101_2 , and 110_2 , respectively. When $A_2A_1A_0$ are equal to 111_2 , all MPROM outputs will be defined to logical 1, as there are connections between the sources of all nMOSFETs of row 7 and the $R_{\text{pull-down}}$. Besides, when the control input pin \overline{EN} is equal to logic 0, the MPROM is enabled to work according to Table 1.7. Nevertheless, when the control input pin \overline{EN} is equal to logic 1, all outputs D_7-D_0 are set to high impedance, i.e., it behaves as an open circuit (simulating an open switch), and we do not have access to it. Additionally, we cannot perform a write operation with the MPROM. In order to perform a read operation of the last memory content, for example, the \overline{EN} must be initially set to logic 0, after the address bus made by A_2-A_0 signals be set to 111_2 . Consequently, the content of this memory location will define the output D_7-D_0 with the byte 11111111_2 .

1.4.2 Programmable Read-Only Memory (PROM)

It is also a non-volatile memory which can be only electrically programmed once by the user (on time programming – OTP) by means of an electronic device named memory programmer. Its internal architecture is similar to the MPROM. It consists of base cells composed of nMOSFETs and fuses connected between their source terminals and the $R_{\text{pull-down}}$, according to Fig. 1.6 (PROM 3×8) [1, 10].

In Fig. 1.6, the V_{pp} is the programming supply voltage, and the **Prog** is the control signal to perform the programming of the PROM [1, 10].

To carry out its programming, initially it is necessary to define that the \overline{EN} be equal to logic 0 to select the V_{pp} , which is defined by the manufacturer, and the address of the memory location in the address bus (A_2-A_0) where we intend to perform the programming. Thenceforth, we must set the data bus (D_7-D_0) with a specific byte. When the **Prog** is set to logic 0, the data bus is stored in this memory location. The pulse width of the V_{pp} depends on the manufacturer's specifications (some milliseconds). This happens because the voltage pulse (V_{pp}) applied in the data bus is capable of flowing sufficient electrical current to burn the fuses, when we set a logical 0 in their contents. When logic 1 is set to the data bus, the electrical

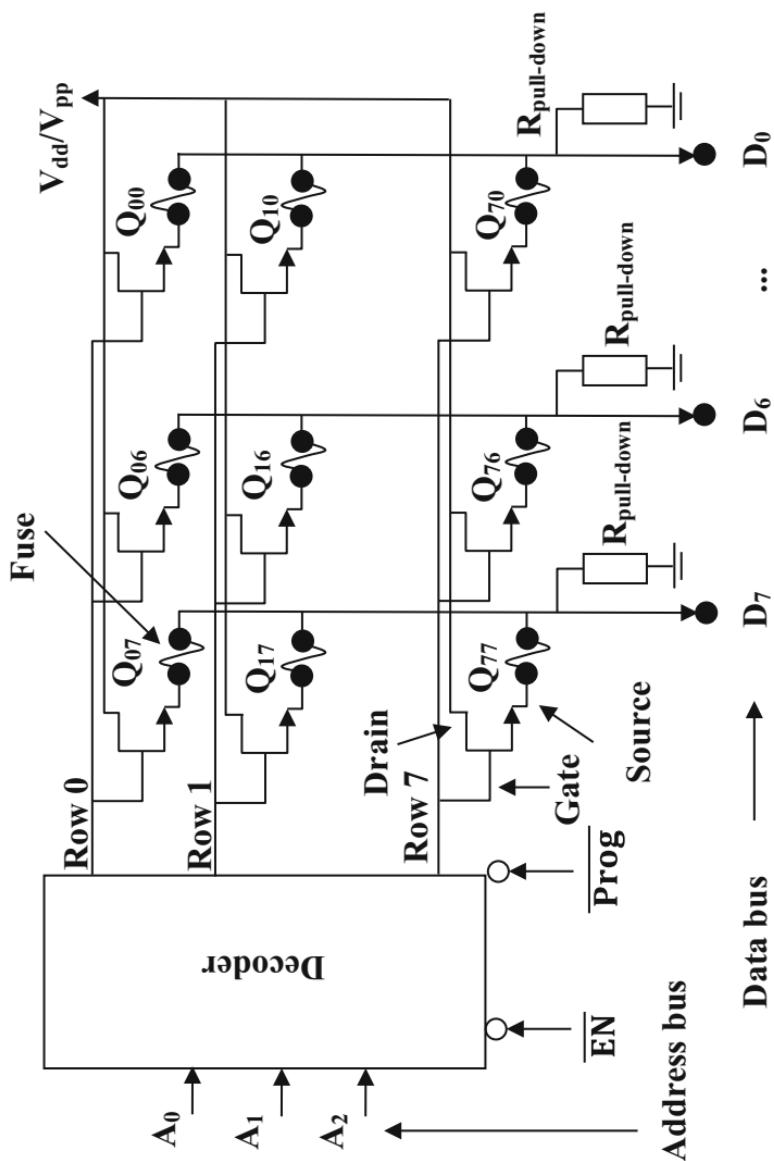


Fig. 1.6 Simplified internal architecture of an 8×8 PROM

current that flows is able to preserve the fuses, and consequently the contents of the PROM are stored with a logic 1 [1, 10].

The read operation of the PROM is similar to the MROM, as previously described [1, 10].

1.4.3 Erasable Programmable ROM (EPROM)

The erasable programmable ROM also has non-volatile characteristics. It can be electrically programmed a lot of times (typically 1000 times, but it depends on the manufacture technology). This memory presents a glass window in its encapsulation, and it can be erased by exposing to the ultraviolet light (UV) (Fig. 1.7). Through the photoelectric effect, the UV light interacts with the MOSFETs' gates, eliminating the charge stored, and it returns to the initial condition of programming (logic 1). The EPROM eraser is the electronic component which is capable of erasing the EPROM [1, 10].

The EPROMs consist of nMOSFETs, which present floating gate terminals, as shown in Fig. 1.8. In this case, the transistor is turned off and each cell is stored with a logic 1. Electron-charge carriers are injected in the floating gate terminal by applying a programming voltage pulse (its value depends on the CMOS technology of the manufacturer), and consequently the transistor is turned on and therefore stores a logic 0 [1, 10].

Fig. 1.7 Example of a commercial EPROM



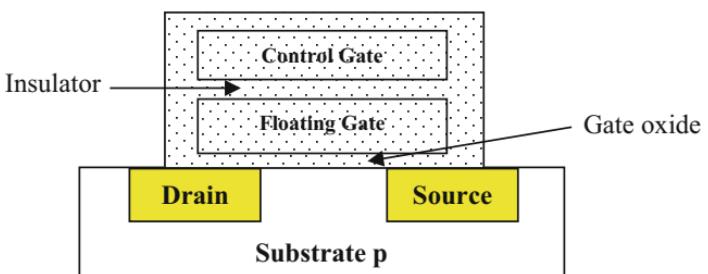


Fig. 1.8 Typical transversal section of a EPROM with floating gate

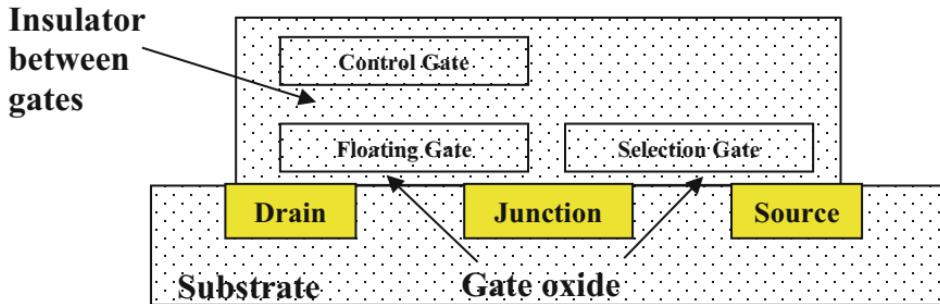


Fig. 1.9 Typical cross section of an EEPROM with floating gate

1.4.4 Electrically Erasable Programmable Read-Only Memory (EEPROM)

This memory has non-volatile characteristics. It can be electrically programmed and erased several times (depending on the manufacture technology). Its core is practically the same as the EPROM, but evolved because it presents a fine oxide region in the drain region, giving it a characteristic of being electrically reprogrammed (Fig. 1.9) [1].

The selection gate is used to define its programming and erasing operations. The programming and erasing operations of the electrical current are very small due to the existence of a small thickness of thin oxide in these devices. Its programming is carried out by applying a voltage pulse (an amplitude that can vary from 18 to 24 V, but it depends on the ICs CMOS technology manufacturer) between the floating gate and drain, which injects charge carriers into the floating gate of the MOSFET. Its erasure can be accomplished by applying a reverse voltage between the open gate and drain, causing the charge to be removed from the transistor gate [1, 10].

The advantage of these memories over the EPROMs is that they can be programmed and erased on the IC board where they were built (in-circuit programming or in-system programming). Consequently, the product development process becomes much more dynamic because it does not need to be removed from the prototype to be programmed in a programmer and erased with ultraviolet light by

using a memory eraser. The disadvantage of its use is that it has a much larger IC area than the one of the EPROMs (it is approximately twice as large) [1,10].

1.4.5 Flash Memory

In the same way as previous ones, the flash memory also presents non-volatile characteristics. Its core resembles EPROMs, but only slightly larger. The only difference is that it has a thinner oxide under the floating gate, making the electrical blanking process possible [1,10].

The main advantages of this type of memory are its high integration density and its reduced access time when compared to the EEPROM [1,10].

1.4.6 Static Random-Access Memory (SRAM)

Static random-access memory is a volatile memory that performs read and write operations. Figure 1.10 shows its internal structure, which it is composed of an address decoder (responsible for accessing a certain memory location that will be read or write), registers (set of D flip-flops implemented with CMOS ICs technology containing 6 MOSFETs: n-type and p-type), a data bus (responsible for defining the data that will be stored in its content), an output buffer (a voltage follower with high electrical current capacity responsible for making available a certain content after a read operation), and finally control signals, such as the read control signal (\overline{RD}) and the write control signal (\overline{WR}). The control signal named (\overline{CS}) has the objective to enable the SRAM to work, when it is logic 0, but it is responsible for disabling it ($\overline{CS} = 1$), i.e., forcing its data bus to operate in the three-state condition, and consequently it is unable to perform the read and write operations. To perform a read operation: $\overline{CS} = \overline{RD} = 0$ and $\overline{WR} = 1$, and to perform a write operation: $\overline{CS} = \overline{WR} = 0$ and $\overline{RD} = 1$. The control signals \overline{WR} and \overline{RD} can never be active simultaneously (the hardware and software designs must guarantee this condition) [1,10].

In Fig. 1.10, I_0, \dots, I_7 are the input data bus, $D_{07}, \dots, D_{00}, \dots, D_{77}, \dots, D_{70}$ are D flip-flops, O_0, \dots, O_7 are the output data bus, $A_2A_1A_0$ is the address bus, and D_7, \dots, D_0 is the data bus.

The memory read operation of this memory is similar to the MROM. Initially, it is necessary to define the value of the address bus (A_2-A_0) to select the memory position whose content will be read and put in the data bus (O_7-O_0). The \overline{CS} must be defined as logic 0 in order for the read operation to be performed, removing the data bus from the three-state. To perform a write operation in the content of a memory position, it is necessary, firstly, to define the address bus of this memory position (A_2-A_0), where you want to store the data. Subsequently, you must also define the data bus (I_7-I_0) with the value that you desire to write in the content of this memory position. Finally, the \overline{WR} control signal must be defined as logic 0. Therefore, the value defined in the data bus will be written in the content of the memory position defined by the address bus [1, 10].

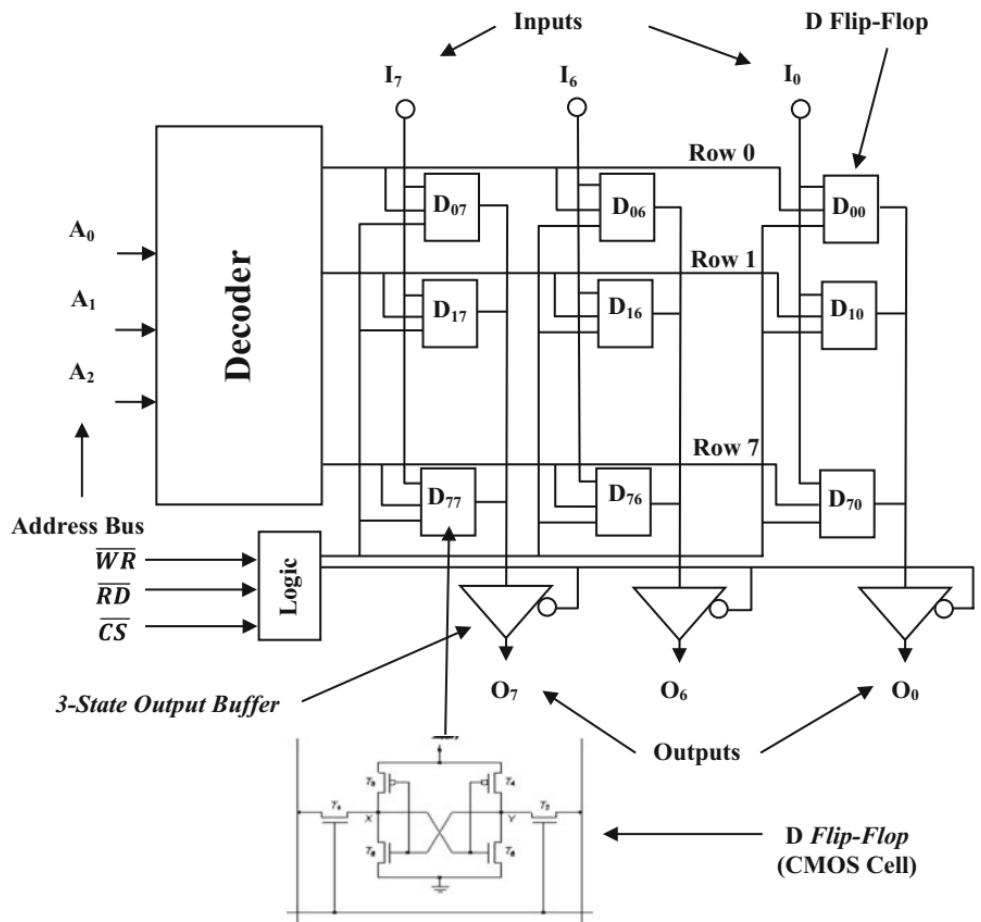


Fig. 1.10 Simplified internal architecture of a static RAM (8x8 SRAM)

1.4.7 Dynamic Random-Access Memory (DRAM)

The DRAM is also a volatile memory. The data is recorded on MOS capacitors, instead of in D flip-flops, as it is done in the SRAM. Its electrical circuit consists of a MOSFET in series with a MOS capacitor, as illustrated in Fig. 1.11. Its internal architecture is similar to the SRAM [1, 10].

In Fig. 1.11, C is the capacitor responsible for storing the data.

The DRAM presents high integration characteristics, and therefore it has a very high data storage capacity. It usually presents a large quantity of addresses' bits. In order to reduce these quantities of addresses' bits, multiplexation techniques are used. This technique consists of sharing at the same address pins (two or more address bits), which are defined at different times, to generate the address of the memory location to be accessed [1, 10].

Fig. 1.11 DRAM basic cell

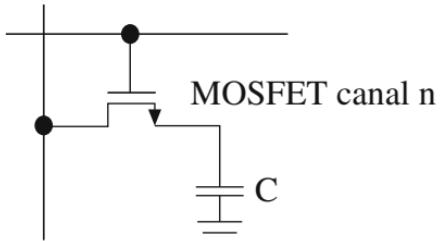
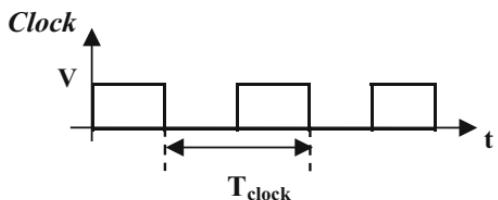


Fig. 1.12 Pulsed square waveform (Clock)



An additional external circuitry, named refresh circuit, is necessary for the DRAM correct operation in order to avoid loss of data in this energy storage element (capacitors). The refresh circuit is responsible for updating the data in the capacitors periodically (this time depends on the CMOS IC technology used) [1, 10].

1.5 Internal Oscillator (Clock)

This electronic circuit is responsible for generating a pulsed square waveform (Fig. 1.12) [2–11].

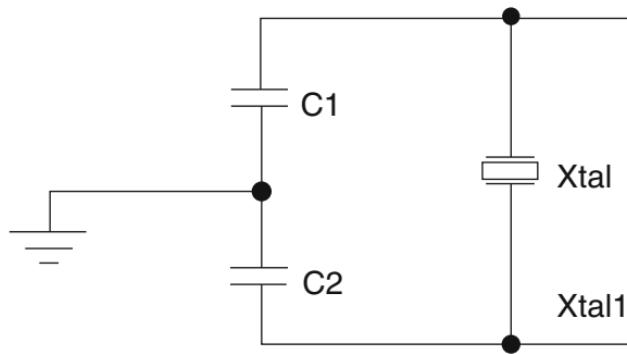
In Fig. 1.12, T_{clock} is the period (periodic time) of the clock and t is the time in seconds.

The frequency of the clock (f_{clock}) is the inverse of the period, and this signal defines the processing speed of computer systems. The period of this pulsed square waveform (T_{clock}) defines the execution (runtime) of the microprocessor's instructions. To illustrate, consider a microcomputer with an internal oscillator (clock) of 2 GHz. Therefore, the f_{clock} is equal to 2 GHz and the T_{clock} is equal to 0.5 nanosecond (ns), calculated by $1/2\text{GHz}$. This means that the computer system executes one instruction every 0.5 ns [2–11].

The crystal, commonly named “xtal,” is the device responsible for generating the clock (pulse square waveform) of the computer system. An example of an external crystal oscillator circuit usually applied in computer systems implemented with 8051 microcontrollers is illustrated in Fig. 1.13 [2–11].

In Fig. 1.13, $C1$ and $C2$ are the capacitors and Xtal1 and Xtal2 are the external crystal oscillator circuit outputs.

Fig. 1.13 Example of an external crystal oscillator circuit commonly used in computer systems implemented with 8051 microcontrollers



1.6 Reset (Initiation) Circuit of a Computer System

All computer systems have an input named “reset” (which is usually a microprocessor/microcontroller pin). When activated by an electric signal of the external circuit, it is responsible for initializing the computer system. This electrical signal of initialization forces the microprocessor of the computer system to run its software from its first instruction. This corresponds to performing an initialization process of the computer system [2–11].

The computer system can be reset in two different ways. The first one occurs when the computer system is powered-on. The second way is through the reset signal, without de-energizing it. As an example, Fig. 1.14 shows the reset circuit of the 8051 microcontroller (Intel) [2–11].

Initially, the capacitor is discharged. When the reset circuit is powered-on (on/off switch is on), the voltage difference over the capacitor is equal to zero, and thus the output voltage of the reset circuit (V_{Reset}), as indicated in Fig. 1.14, is equal to the supply voltage (V_{dd}), which generates a reset signal for the microprocessor/microcontroller of the computer system. After five time constants ($5\tau = 5 \cdot 10\mu F \cdot 10K\Omega = 0.5$ s, in this case), the capacitor is charged with the V_{dd} value, causing the V_{Reset} to be equal to zero, finishing the reset process of the computer system. Besides, when the push-button switch is activated, regarding the computer system is powered-on, the capacitor is discharged through it and a new reset signal is generated, causing the microprocessor/microcontroller to start running its software from its beginning by running the first instruction that composes its software (initialization process of the computer system) [2–11].

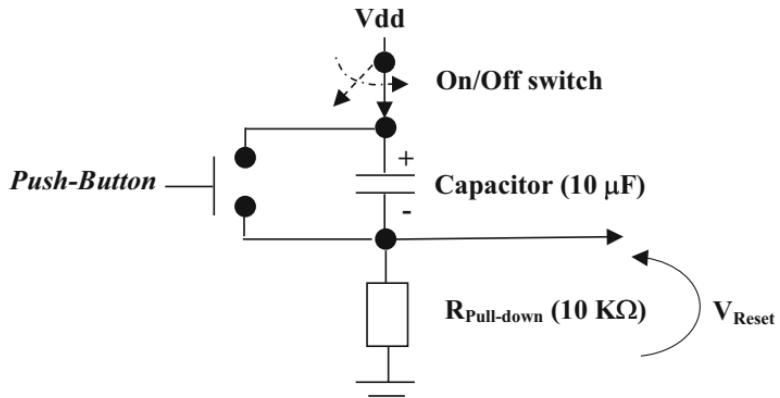


Fig. 1.14 Reset circuit of a computer system implemented with 8051 microcontrollers

1.7 The Input Devices and Their Associated Interfaces for Computer Systems

The input devices and their associated interfaces are responsible for providing data/information from the external environment to the computer systems. This data/information will be necessary to control a specific process to be controlled. Some examples of input devices are switch, sensor, keyboard, mouser, hard drive, pen driver, etc. Commonly, the input devices and their associated interfaces are responsible for transforming a physical quantity of nature (temperature, pressure, light intensity, etc.) into an electrical magnitude, such as electrical voltage, current, resistance, etc. [3, 11].

1.7.1 Mechanical Switches

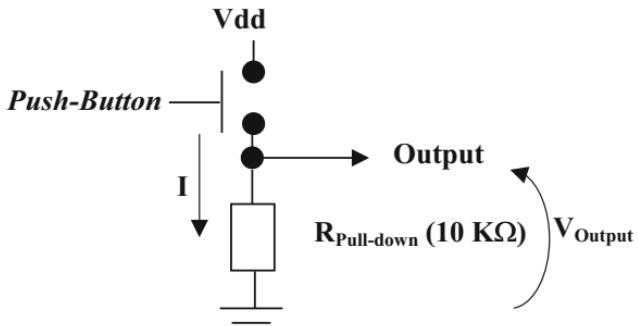
Mechanical switches are mechanical input devices commonly used on keyboards or to define open-circuit or short-circuit conditions, in order to indicate, for instance, the condition of an open or closed door, regarding a residential alarm system, etc. [10].

The computer systems usually have an input interface (generally named as input ports) to receive data/information from the external environment. An example that we can mention is the push-button and the associated input interface, which can be connected to these computer systems as shown in Fig. 1.15. The on/off switch can also be used with this type of circuit [3, 11].

In Fig. 1.15, I is the electrical current and V_{Output} is the output voltage of the circuit.

When the switch is off, the circuit output presents an electrical voltage of zero volts (logic 0) because there is no electrical current flowing in this circuit and consequently there is no voltage drop on the resistor ($V_{Output} = R_{pull-down} \cdot I$)

Fig. 1.15 The push-button and associated input interface



$= R_{\text{pull-down}} \cdot 0 = 0 \text{ V}$). When the switch is activated, there is an electrical current flowing in the circuit, given by the supply voltage value (V_{dd}) over the resistance value ($R_{\text{pull-down}}$). As the switch is activated, the voltage drop over it is practically equal to zero, and consequently the output voltage of the circuit is equal to the supply voltage (V_{dd} , logic 1). Observe that this circuit is capable of transforming the physical condition of the switch (on or off) into the logical levels (0 or 1), depending on its condition. Therefore, the role of the software designer is to relate the physical condition of the switch with the logic levels generated by the circuit in order to design its computer system. Regarding the hardware/software designers, this approach is valid for any other sensors/transducers used in the computer systems [3, 11].

1.7.2 Matrix Keyboard

A keyboard containing 16 push-button switches can be implemented according to Fig. 1.15, as explained in the previous section. In this case, we must use 16 push-button switches, 16 inputs of the computer system, and 16 pull-down resistances. However, this type of architecture for keyboards is rarely used in computer systems. In order to reduce the cost of the implementation of matrix keyboards, the “scanning technique” for matrix keyboard is used. Figure 1.16 illustrates an example of a related piece of hardware for matrix keyboards to apply this methodology (in this case a 4×4 matrix keyboard) [3, 11].

In Fig. 1.16, C_0, \dots, C_3 are the columns, L_0, \dots, L_3 are the rows, D_0, \dots, D_3 are diodes: used to avoid a short circuit when two or more switches belonging to different columns are activated simultaneously, $CH_{x,y}$ are the push-button switches ($0 \leq x \leq 3$ and $0 \leq y \leq 3$), and R_z are the pull-down resistances ($0 \leq z \leq 3$) [3, 11].

Observe that in this case, we must use the same 16 push-button switches, now only 4 inputs (L_0, \dots, L_3) and 4 outputs (C_0, \dots, C_3), i.e., a total of 8 inputs/outputs of the computer system, and only 4 pull-down resistances. Therefore, we have a reduction of 50% in terms of number of inputs/outputs of the computer system and 75% of the quantity of the pull-down resistance $\{=[1-(4/16)].100\%\}$ [3, 11].

The scanning technique to decode this keyboard is performed as follows:

(1) Firstly, we must define 4 outputs (C_0, \dots, C_3) and 4 inputs (L_0, \dots, L_3) of the computer system. Based on these 4 outputs and 4 inputs, we must define a byte, such

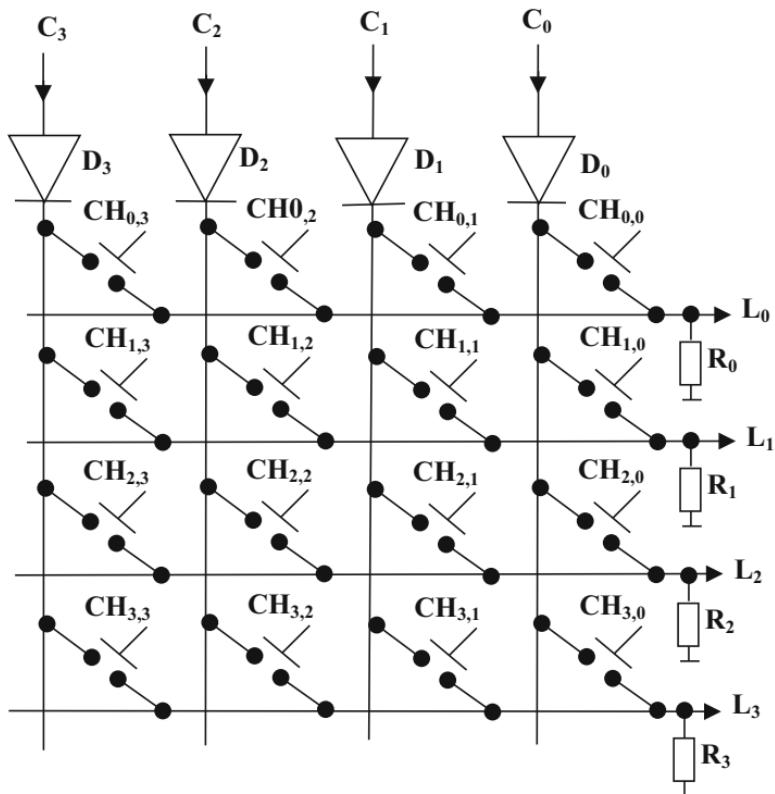


Fig. 1.16 A example of a 4×4 matrix keyboard and its associated interface with push-button switches

as $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)$, where the 4 most significant bits (superior nibble) are composed by the inputs (L_3 , L_2 , L_1 , and L_0) and the 4 least significant bits (inferior nibble) are composed of the outputs (C_3 , C_2 , C_1 , and C_0).

(2) Afterwards, we must activate the first column ($C_0 = 1$) and deactivate the other columns ($C_0 = 1$, $C_1 = 0$, $C_2 = 0$, and $C_3 = 0$); then we must read the lines (L_0 , ..., L_3), which will define a byte. This byte must then be analyzed. To illustrate, if no row is in logic 1, then there are no activated switches, regarding those belonging to column 0 ($CH_{0,0}$, $CH_{1,0}$, $CH_{2,0}$ or $CH_{3,0}$), i.e., $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)_2 = 00000001_2$. Besides, for example, if line L_0 goes to logic 1, then the $CH_{0,0}$ key has been activated, composing the byte $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)_2 = 00010001_2 = 11_{16} = 11h$. Similarly, if line L_1 goes to logic 1, then the $CH_{1,0}$ key has been activated, defining the byte $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)_2 = 00100001_2 = 21_{16} = 21h$. If line L_2 goes to logic 1, then the $CH_{2,0}$ switch has been activated, composing the byte $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)_2 = 01000001_2 = 41_{16} = 41h$. Finally, if line L_3 goes to logic 1, then the $CH_{3,0}$ switch has been activated. Furthermore, if the lines L_0 and L_1 become logic 1, this means that the $CH_{0,0}$ and $CH_{1,0}$ were activated at the same time, defining the byte $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)_2 = 00110001_2 = 31_{16} = 31h$. This procedure is able to obtain any combination of the activated switches concerning this

activated column (C_0). This procedure must be repeated for other columns. Therefore, if we use this scanning technique, we can scan and detect the activation of any switch of this keyboard. In the case of detecting the activations of two or more switches of a keyboard which do not belong to the same column, one possibility is to perform the scan for all columns, storing the bytes generated by each column in different memory positions. Subsequently, they must be analyzed by using a procedure with a specific strategy in order to detect the activation of these different switches. For instance, if the first and second bytes generated by the two first scanning of the keyboard are equal to $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)_2 = 00010001_2 = 11_{16} = 11h$ and $(L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0)_2 = 00100010_2 = 22_{16} = 22h$, the $CH_{0,0}$ and $CH_{0,1}$ were activated simultaneously. Table 1.8 shows some bytes generated regarding the activation of one or more switches of a 4×4 matrix keyboard [3, 11].

In order to scan a matrix keyboard, a predefined time base is regularly used with a timer/counter block (the typical value of the scan time is 1 ms for each column,

Table 1.8 Bytes generated by the scanning technique for matrix 4×4 matrix keyboards

$L_3 L_2 L_1 L_0$	$C_3 C_2 C_1 C_0$	<i>Byte in hexadecimal ($L_3 L_2 L_1 L_0 C_3 C_2 C_1 C_0$)</i>	Activated key
0000	0001	01	None
0001	0001	11	$CH_{0,0}$
0010	0001	21	$CH_{1,0}$
0100	0001	41	$CH_{2,0}$
1000	0001	81	$CH_{3,0}$
0011	0001	31	$CH_{0,0}$ and $CH_{1,0}$
0000	0010	02	None
0001	0010	12	$CH_{0,1}$
0010	0010	22	$CH_{1,1}$
0100	0010	42	$CH_{2,1}$
1000	0010	82	$CH_{3,1}$
0111	0010	32	$CH_{0,1}$, $CH_{1,1}$, and $CH_{2,1}$
0000	0100	04	None
0001	0100	14	$CH_{0,2}$
0010	0100	24	$CH_{1,2}$
0100	0100	44	$CH_{2,2}$
1000	0100	84	$CH_{3,2}$
1111	0100	F4	$CH_{0,2}$, $CH_{1,2}$, $CH_{2,2}$, and $CH_{3,2}$
0000	1000	08	None
0001	1000	18	$CH_{0,3}$
0010	1000	28	$CH_{1,3}$
0100	1000	48	$CH_{2,3}$
1000	1000	88	$CH_{3,3}$
0110	1000	68	$CH_{1,3}$ and $CH_{2,3}$

totaling a total scan time of 4 ms) for a 16-key 4×4 keypad. The same strategy can be used to scan matrix keyboards with different number of switches [3, 11].

1.7.3 Sensors and Their Associated Interfaces

The sensors are fundamental devices for automation in different areas of the electrical engineering (robotics, industrial, building and residential, car automations, etc.). Currently there are a great variety of sensors, such as sensors of temperature, pressure, humidity, presence, speed, capacitive, inductive, etc. Some examples of sensors and their associated interfaces commonly used in computer systems are presented below [3, 11].

I. Light-Dependent Resistor (LDR): It is a resistor that varies its electrical resistance as a function of the intensity of electromagnetic radiation of the visible spectrum (Fig. 1.17) [10].

The LDRs are usually implemented with cadmium sulfide (CdS) or cadmium selenide (CdSe), and their resistances decrease as a function of the light intensity (low when there is light and very high when is dark). An example of the LDR applications is public lighting control [11].

A simple interface containing an LDR which can be used in computer systems is shown in Fig. 1.18 [11].

The output voltage (V_{output}) of this interface will be logic 1, when the LDR is under the influence of the light. This can be justified because its electrical resistance becomes very low (typically 100Ω), and consequently its voltage drop is practically negligible in relation to that observed in the potentiometer of $100k\Omega$ (practically V_{dd}). When the LDR is in a dark environment, its resistance becomes very high (typically some $M\Omega$) and consequently its voltage drop becomes much higher (practically V_{dd}) than that measured in the potentiometer. Therefore, the V_{output} goes to logic 0 (V_{output} is equal to V_{dd} subtracted by the voltage drop of the LDR). The function of the potentiometer is to set the level of brightness to be controlled [11].

Fig. 1.17 Simplified photograph of an LDR

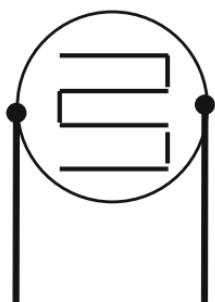


Fig. 1.18 The LDR and its associated interface

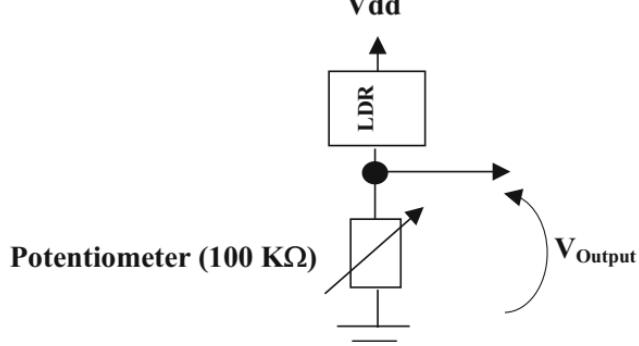
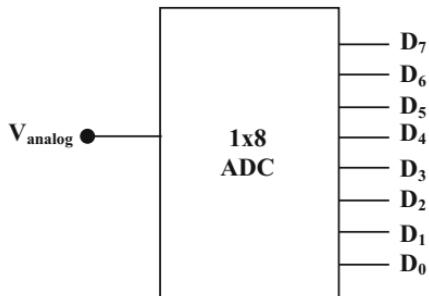


Fig. 1.19 Example of a 1×8 DAC



1.7.4 Analog-to-Digital Converter

The analog-to-digital converter (ADC) is an integrated circuit (IC) capable of transforming an analog signal into a digital signal composed of n (integer number) bits in its output. To illustrate, consider an ADC that presents one analog input and its output is composed of 8 bits, 1×8 ADC (Fig. 1.19) [11].

Consider that the analogical signal voltage (V_{analog}) can vary from 0 to 5 V. When the V_{analog} is equal to zero, ideally the 1×8 ADC will convert this input signal to a digital signal (D_7-D_0) to 00000000_2 (minimum value that can be represented on the output). In the same way, when the V_{analog} is equal to 5 V, the byte (D_7-D_0) in the circuit output will be equal to 11111111_2 (maximum value that can be represented in the output). Since 1×8 DAC has 8 bits in its output, there are 256 (2^8) possible binary combinations which can be represented in the output (D_7-D_0) of this circuit. Therefore, for every bit of variation in the output, it will correspond to a variation in the input voltage of 19.53 mV (maximum $V_{\text{analog}} - \text{minimum } V_{\text{analog}}/2^n = (5-0)V/2^8 = 5 V/256 = 19.53$ mV (Table 1.9) [11].

Table 1.9 The V_{analog} and the output byte of a 1×8 DAC

#	V_{analog} (V)	$(D_7-D_0)_2$	$(D_7-D_0)_{16}$
1	0	00000000	0
2	19.53×10^{-3}	00000001	1
3	39.06×10^{-3}	00000010	2
4	58.59×10^{-3}	00000011	3
:			
128	2.5	01111111	7F
:			
255	4.9805	11111110	FE
256	5	11111111	FF

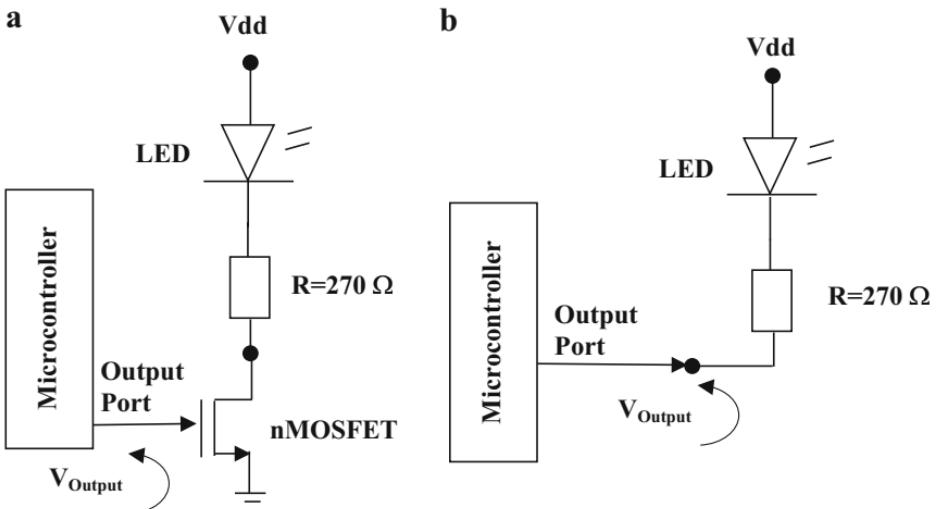


Fig. 1.20 Examples of two different interfaces for LEDs: (a) by using positive logic and (b) by using negative logic

1.8 The Output Devices and Their Associated Interfaces for Computer Systems

The output devices (actuators) and their associated interfaces are responsible for providing a response of computer systems to the external environment in which it is operating. Some output devices used in computer systems are LEDs, 7-segment displays, beeps, buzzers, liquid crystal displays, relays to drive motors, etc. [3,11].

1.8.1 The LED and Their Associated Interfaces

Figure 1.20 illustrates two different examples of interfaces for LEDs [3, 10].

In the first one (Fig. 1.20a), the LED is activated (positive logic) when the microcontroller defines logic 1 in its output port through a piece of software which

was previously implemented by a software programmer. This interface requires the use of a transistor whose purpose is to supply the electric current appropriate that must flow by the LED to generate a specific brightness. This happens because the output electrical current of an output port of the microcontroller is usually not capable of supplying such electrical current level to an active LED (some milliamperes). In the second interface, the LED is activated when the microcontroller defines the logic 0 (negative logic) in its output port (Fig. 1.20b). This type of interface is widely used because the absorbing capacity of the electrical current by an output port of a microcontroller is usually higher than the one that it can drive out regarding a specific load. Besides, this interface type is able to reduce the cost of the MOSFETs used in the previous interface (Fig. 1.20a) [3, 11].

Since the voltage drop (V_{LED}) of a commercial LED is approximately 2.3 V (typical) in order to produce a suitable brightness in well-lit environments, its electrical current (I_{LED}) must be approximately equal to 10 mA. Therefore, the value of the electrical resistance (R) that must be connected in series with the LED is given by $R = (V_{dd} - V_{LED})/I_{LED}$. For example, if the V_{dd} is equal to 5 V, the R will be equal to $(5 \text{ V} - 2.3 \text{ V})/10 \text{ mA} = 270\Omega$ [3, 11].

1.8.2 Bar LEDs and Its Associated Interface

A set of LEDs composes the so-called LEDs bar. To illustrate, consider a LEDs bar containing a set of 8 LEDs. You can use the same interfaces used in the previous section (Sect. 1.8.1), regarding the positive or negative logics, to implement this interface. Figure 1.21 illustrates an example of an interface of LEDs bar containing 8 LEDs, regarding the negative logic [3,11].

Table 1.10 shows some examples of bytes that can be defined by the microcontroller in their output bits of the port 0 to activate the LEDs bar regarding different combinations [3, 11].

1.8.3 7-Segment Display

A 7-segment LEDs display consists of 7-segment-shaped LEDs, defined as segments a, b, c, d, e, f, and g and a circular LED, called dp (Fig. 1.22). It can be built as a common cathode or a common anode. When the display is a common cathode, it means that all the cathodes of each LED are interconnected with each other, so that each LED can be activated when logic 1 is applied and deactivated when logic 0 is applied. When the display is a common anode, it means that all the anodes of each LED are interconnected with each other, and thus each LED can be activated when logic 0 is applied and deactivated when logic 1 is applied. It is manufactured to be commonly used in electronic machines, whose communication with humans is important, such as the calculators, plastic injection machines, etc. [3, 11].

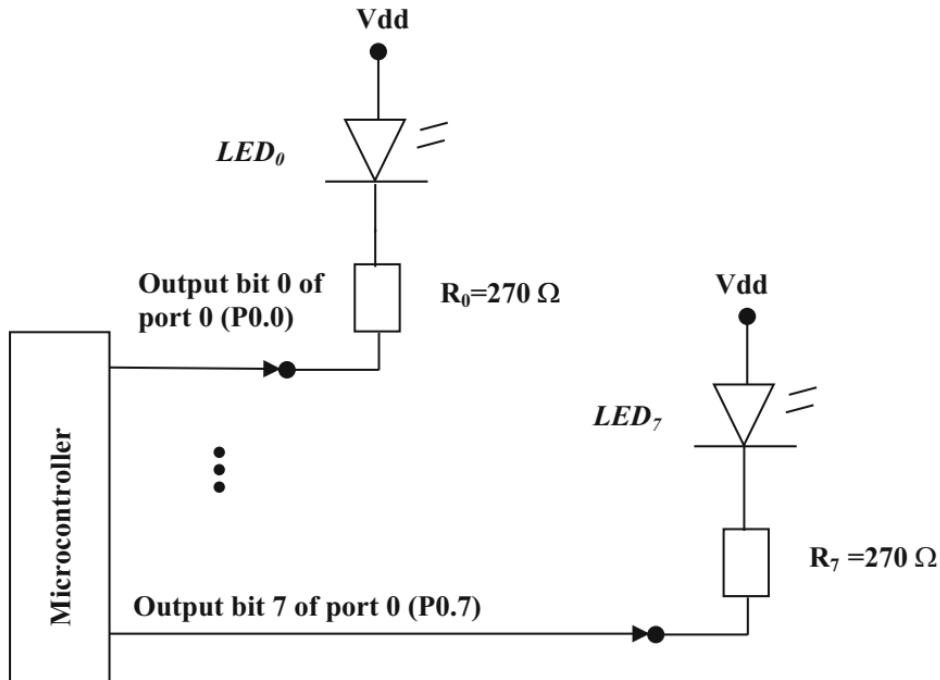


Fig. 1.21 An example of a LEDs bar and its corresponding interface (negative logic)

Table 1.10 Some bytes defined by the microcontroller and the electrical conditions of the LEDs bar

$(P0.7, \dots, P0.0)_2$	$(P0.7, \dots, P0.0)_h$	LED_7, \dots, LED_0
00000000	00 _h	All activated
00000001	01 _h	Only LED_0 is not activated
10000000	80 _h	Only LED_7 is not activated
01010101	55 _h	Only the LEDs with odd indices are activated
00001111	0F _h	Only the four most significant LEDs are activated
11110000	F0 _h	Only the four least significant LEDs are activated
11111111	FF _h	All deactivated

Table 1.11 exemplifies some bytes that define the input of the common anode 7-segment display in order to define some numbers, letters of the alphabet, etc. (7-segment codes) [3, 11].

You can build a display with several 7-segment displays to write some information, such as the hours of a clock, a message, machine conditions, etc. Figure 1.23 illustrates a piece of hardware for “n” (n integer) common anode 7-segment displays [3, 11].

In Fig. 1.23, D_0, \dots, D_7 are the 7-segment displays, PNP_1, \dots, PNP_n (n is an integer number) are bipolar transistors, Y_0, \dots, Y_n are the output ports of a microcontroller which are connected to the corresponding bits of Port 1 ($P1 = P1.0, \dots, P1.7$), and dp, \dots, a are the data inputs of the LEDs of the

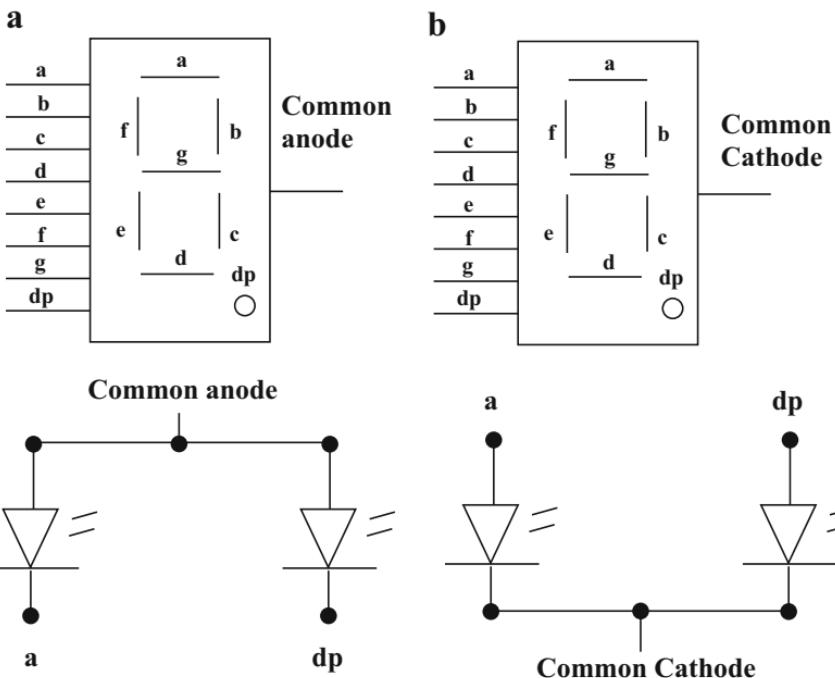


Fig. 1.22 Examples of 7-segment displays: common anode (a) and common cathode (b)

Table 1.11 Some examples of bytes (0, on, and 1, off), named 7-segment code, defined for the inputs of the 7-segment display (a, b, c, d, e, f, g, and dp)

(dp g f e d c b a) ₂	(dp g f e d c b a) _h	Character in 7-segment display
00000000	00 _h	Number “8.” (all activated)
11111100	F9 _h	Number “1”
10100100	A4 _h	Number “2”
10000011	83 _h	Letter “b”
10001000	88 _h	Letter “A”
11000110	86 _h	Letter “C”
11111111	FF _h	All deactivated
01111111	7F _h	Only “dp LED” is activated (“.”)

7-segment displays, which are connected to the corresponding bits of the port 0 ($P0 = P0.0, \dots, P0.7$) [3,11].

All data inputs (dp, ..., a) of the 7-segment displays must be connected to Port 0 of the microcontroller. This port (P0) is responsible for defining the 7-segment code which will be written in the displays. Port 1 (P1) is responsible for selecting one 7-segment display at a time, e.g., when the P1.0 (Y_0) is activated in logic 0 (negative logic), the PNP₀ operates as a short circuit, and consequently the 7-segment display D₀ is powered-on. When the P1.1 (Y_0) is activated in logic 0, the PNP₁ operates as a short circuit, and consequently the 7-segment display D₁ is powered-on and so

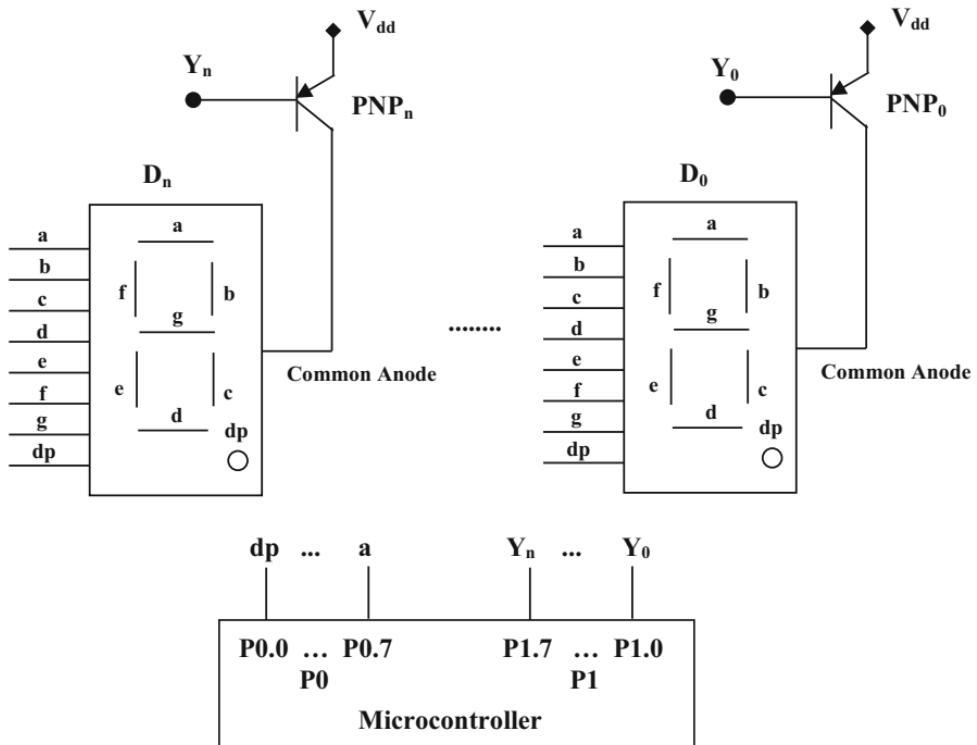


Fig. 1.23 Example of a piece of hardware containing “n” common anode 7-segment displays and a microcontroller for defining the output ports to control it

on. The procedure of writing a 7-segment code in the displays must be performed one at a time. Firstly, we must select the 7-segment display on which we desire to write a data, and afterwards we must define the 7-segment code (byte) in the $P0$; consequently the data is written in this specific display. In order to write a message in all 7-segment displays, we must initially define the 7-segment display (D_0 , e.g., by defining $P1.0 = Y_0$ equal to logic 0), and then we must define the 7-segment code in the $P0$ (dp, ..., a). Subsequently, this same procedure must be performed for the other displays. We usually use a Timer to generate interruptions every 2 ms, for example, to the microcontroller. Each time that the microcontroller is interrupted, it performs a scan in the 7-segment displays, i.e., activating one specific display and writing a 7-segment code on it. If the display is composed of four 7-segment displays, in 8 ms, a message is written on it. It is not necessary that the microcontroller be performing writing operations on the displays all the time because the retention time of the retinas of our eyes is much greater than 2 ms. If the time between interruptions of the Timer is higher than 2 ms, the brightness of the displays can be reduced and it can flicker. This procedure to write information in 7-segment displays by using a Timer is called “display scanning technique,” which is similar to that already previously described for the keyboard. Usually, when a scan is performed on the displays, the keyboard scanning is also performed [3,11].

1.8.4 Liquid Crystal Display (LCD)

The liquid crystal display consists of two transparent blades, composed of cells. Between the cells there is a liquid, which is polarized by light (liquid crystal). It is electrically controlled by electrical voltage through two electrodes, which are able to change the orientation of its molecules, consequently allowing the passage of light and thus generating an image on this display [3,11].

The commercial LCD is usually built with a microcontroller associated to it, which is responsible for interacting with other microcontrollers and controlling it. LCDs are mainly used in wristwatches, calculators, radios, televisions, etc. [3,11].

An example of an LCD containing 1 row and 16 columns (LCD 1×16) is shown in Fig. 1.24 [3,11].

In Fig. 1.24, the V_{CC} pin must be connected to the supply voltage (+); the V_{EE} pin (contrast adjustment) is an input that must be connected to a potentiometer to adjust the luminous intensity (brightness) of the LCD (backlight); the V_{SS} pin must be connected to the ground (-) of the power supply; D_7-D_0 is the data bus, by which the LCD commands are defined; and the American Standard Code for Information Interchange (ASCII) codes are written on the LCD. \overline{EN} is the enable input (when a falling edge is defined, it enables write and read operations of the internal memory of the LCD). R/W is the input control signal that enables a read operation ($R/W=1$) and write operation ($R/W=0$) in the internal memory of the LCD ($R/W=1$), and $\overline{R/S}$ is an input control signal that can define a command for the LCD ($\overline{R/S}=0$) (configurations, such as to move the cursor to the left position, clear a message, flash the cursor, etc.) or a write/read operation of a character; it will either perform a one-character write operation on the LCD or an operation in/of a memory location of

Memory Address (row)	C ₀	C ₁	C ₂	C ₃	C ₄	C ₅	C ₆	C ₇	C ₈	C ₉	C ₁₀	C ₁₁	C ₁₂	C ₁₃	C ₁₄	C ₁₅
	0 _h	1 _h	2 _h	3 _h	4 _h	5 _h	6 _h	7 _h	8 _h	9 _h	A _h	B _h	C _h	D _h	E _h	F _h

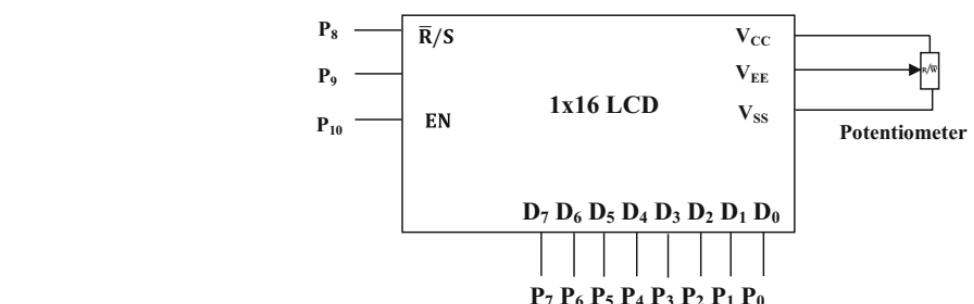


Fig. 1.24 The memory address of each character that can be accessed in the 1×16 LCD and its pinout

Table 1.12 Table of commands and read/write operations of the 2×16 LCD 44780

4	5	14	13	12	11	10	9	8	7	Pins
R/S	R/W	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	Control signals and data bus/command
0	0	0	0	0	0	0	0	0	1	Clear display
0	0	0	0	0	0	0	0	0	X	Initial cursor position (0 _h)
0	0	0	0	0	0	0	0	ID	S	Cursor direction
0	0	0	0	0	0	0	D	C	B	Enable (display/cursor)
0	0	0	0	0	1	SC	RL	X	X	Move cursor
0	0	0	0	1	DL	N	F	X	X	Memory size
0	0	0	1	A	A	A	A	A	A	Move cursor into the memory
0	0	1	A	A	A	A	A	A	A	Move cursor
0	1	BF	X	X	X	X	X	X	X	Busy flag
1	0	D	D	D	D	D	D	D	D	Write operation
1	1	D	D	D	D	D	D	D	D	Read operation

the LCD. For example, Table 1.12 presents how to configure/program and write/read a character of the 2×16 LCD 44780 [3,11].

In Table 1.12, X can be logic 0 or 1, ID is responsible for changing the cursor to the next column after a write operation in a memory location, S is able to shift the typed character for the next column, D is responsible for turning on the display, C is capable of activating the cursor, B is able to flicker the cursor, SC is responsible for moving a character to other position, RL is able to set the direction of writing, DL is responsible for defining the length of the data (0: 4 characters and 1: 8 characters), N is able to set the number of display lines (0: 1 line and 1: two lines), F is responsible for setting the characters' quantity of dots (0: 5×7 and 1: 5×10), BF is an output and indicates when it is busy to do a new operation or it is in processing, A is the memory address of the different LCD/CGRAM characters (character generator RAM: where the user can create characters or symbols), and D is the data/character of the ASCII (Table 1.13), which desires to write on the LCD or to read from the LCD [3, 11].

To illustrate, based on Table 1.13, the number zero in ASCII code is defined by the composition of the bits given by the crossing of column 3 (4 most significant bits) with line 0 (4 least significant bits), i.e., given by the ASCII byte $00110000_2 = 30_{h}$. Similarly, the letter S (uppercase) is defined by the composition of the bits given by the crossing of column 5 with line 3, that is, given by the byte in ASCII $010100112 = 53_{h}$, and so on.

Table 1.13 The ASCII codes of characters of a typical LCD

Upper 4 bits Lower 4 bits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0 0000	CG RAM (1)				ⒶⓐP	ⒷⓐP					—	タ	ミ	シ	ピ	
1 0001	CG RAM (2)				! 1 A Q a q						。	ア	チ	カ	シ	q
2 0010	CG RAM (3)				" 2 B R b r						「	イ	リ	×	ビ	8
3 0011	CG RAM (4)				# 3 C S c s						」	ウ	テ	モ	ス	ω
4 0100	CG RAM (5)				\$ 4 D T d t						、	エ	ト	ト	ム	Ω
5 0101	CG RAM (6)				% 5 E U e u						=	オ	ナ	1	6	0
6 0110	CG RAM (7)				& 6 F V f v						ヲ	カ	ニ	ヨ	ρ	Σ
7 0111	CG RAM (8)				* 7 G W g w						フ	キ	ヌ	ラ	g	π
8 1000	CG RAM (1)				(8 H X h x						イ	ク	ネ	リ	ゞ	☒
9 1001	CG RAM (2)) 9 I Y i y						ウ	ケ	ノ	ル	”	γ
A 1010	CG RAM (3)				* 1 J Z j z						エ	コ	ル	レ	j	‡
B 1011	CG RAM (4)				+ ; K C k <						オ	サ	ヒ	ロ	*	元
C 1100	CG RAM (5)				, < L ¥ 1						ヤ	シ	フ	ワ	Φ	円
D 1101	CG RAM (6)				- = M] m)						ユ	ス	ヘ	ン	モ	÷
E 1110	CG RAM (7)				. > N ^ n +						ヨ	セ	ホ	・	ん	ē
F 1111	CG RAM (8)				/ ? O _ o +						ツ	ソ	マ	?	ö	█

1.8.5 Pulse Width Modulation (PWM)

This technique is used to apply pulses (square wave: part of time is active/on and the rest of time is deactivated/off) to an output interface which usually controls the operation conditions of a specific load, such as the velocity of the direct current (DC) motors, brightness of LED and lamps, frequency of a beep, etc. To illustrate,

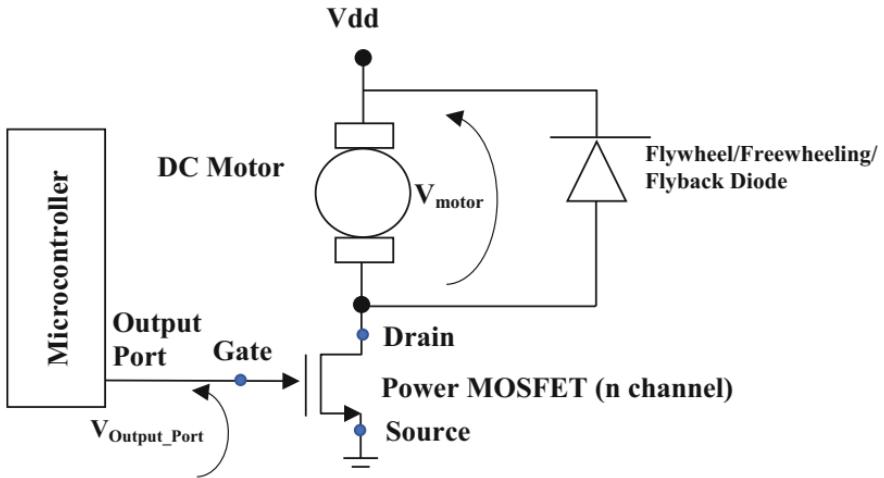


Fig. 1.25 Speed control interface of DC motors

consider an output interface which controls the speed of a direct current (DC) motor, as shown in Fig. 1.25 [3, 11].

In Fig. 1.25, $V_{\text{output_Port}}$ is the port output voltage of the microcontroller and V_{motor} is the motor potential difference [3, 11].

This technique is based on applying a pulse with fixed period (T) in the output port of the microcontroller, which is connected to the interface input (gate of the Power MOSFET, n channel). If logic 1 is defined during all this period (T) by output port (duty cycle of the PWM is equal to 100%), the Power n channel MOSFET will be active (operating as short circuit) during all T , and the value of the supply voltage (V_{dd}) will be practically applied over the DC motor. As the speed of the DC motor (linear) depends on the average voltage (v_a) applied to it, its velocity will practically be equal to the nominal (maximum) speed. If the logic 1 is applied only during the half of the T (duty cycle of the PWM is equal to 50%), the Power n channel MOSFET will be turned on (operating as short circuit) only during half of the T . In this case, the average voltage applied in the DC motor will be practically the half of the V_{dd} , and therefore, its velocity will be practically equal to the middle of the nominal (maximum) velocity and so on. The freewheel (flywheel or flyback) diode must be used in parallel with the DC motor because the DC motor presents a strongly inductive load. When it is powered-off, this inductive load of the DC motor produces a reverse voltage over it, trying to keep fixed the drain current of the Power MOSFET. Because of this, the Power MOSFET can be damaged if this reverse voltage is not discharge by the freewheel (flywheel or flyback) diode [3, 10].

Figure 1.26 illustrates some waveforms of the PWM that the microcontroller can generate to control a specific load (Fig. 1.26a, duty cycle of 100%, and Fig. 1.26b, duty cycle of 50%) [3, 10].

The same procedure can be done to generate different duty cycles and consequently make a speed control that can vary from the stopped motor condition until

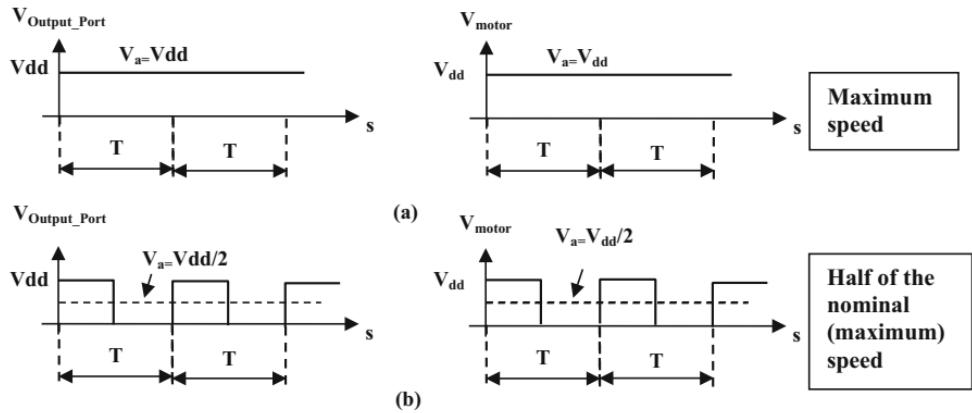


Fig. 1.26 Examples of two different PWMs: duty cycle = 100% (a) and duty cycle = 50% (b) to control the speed of the DC motor

reaching its maximum speed. Similarly, a light control of a lamp can be done using the PWM, as it was done to control the speed of the DC motor [3, 11].

The Timer/Counter of a microcontroller can be also programmed to define the duty cycle of the pulse (time of the square waveform signal is activated and deactivated) to control the energization of a load and consequently to control the speed of motors, the brightness of lamps, etc. [3, 11].

1.8.6 Activation of DC and Alternating Current (AC) Loads

The DC loads can be controlled by using the same interface illustrated in Fig. 1.23. One of the ways to control the operation conditions of AC loads is by using the relays, as shown in Fig. 1.27 [3, 11].

By applying logic 1 in the output port of the microcontroller, the Power nMOSFET is activated, energizing the coil that is connected to its drain. Consequently, the magnetic field generated by it is able to close the contact of the switch, which results in the energization of the load with the AC power supply. The AC load can be an AC motor, a lamp, etc. This same concept can be applied to a three-phase AC motor, in which each phase can be controlled individually [3, 11].

Another interface to be used with a microcontroller is that implemented by using optocouplers and a silicon-controlled rectifier (SCR), as shown in Fig. 1.28 [3, 11].

In Fig. 1.28, $V_{\text{output_Port}}$ is the port output voltage of the microcontroller and $V_{\text{AC_load}}$ is the AC motor potential difference, which is equal to the AC power supply when turned on because the voltage drop in the SCR is practically equal to zero, R_1 is the resistance to bias the diode of the optocoupler, R_2 is the resistance to bias the bipolar transistor of the optocoupler, and R_G is the resistance to bias the SCR [3, 11].

In the same way, by applying logic 1 in the output port of the microcontroller, the nMOSFET is triggered (simulates a short circuit), activating the LED of the

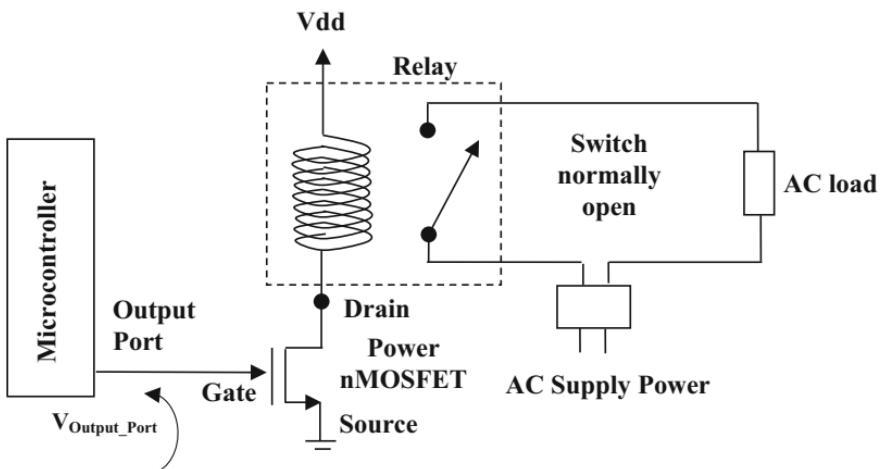


Fig. 1.27 Interface for driving AC loads

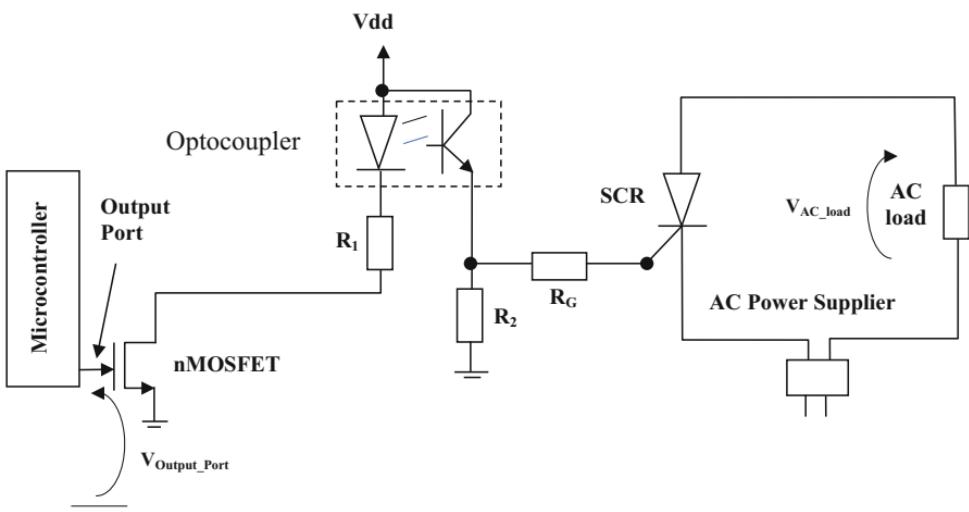


Fig. 1.28 Interface for driving AC loads

optocoupler that is connected to its drain, resulting in the activation of the bipolar transistor of the optocoupler. In this way, a voltage is defined in the gate of the SCR, causing the trigger of the SCR, which results in the power-on of the load [3,11].

1.8.7 The Digital-to-Analog Converter (DAC)

The DAC is an integrated circuit that transforms a digital data (a byte for example) into an analog voltage signal. To illustrate, consider a DAC (Fig. 1.29) with three (3) bits (E_2, E_1, E_0) that define the input digital data and another input pin, named

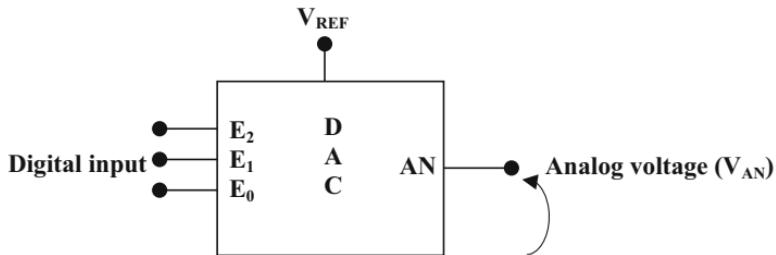


Fig. 1.29 Example of a DAC of 3 input digital data and 1 analog output

Table 1.14 Binary combinations of the inputs and their corresponding analog values in DAC output, considering a V_{REF} of 7 V

Digital inputs			Analog output
E_2	E_1	E_0	V_{AN} (V)
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

reference voltage (V_{REF}), which is responsible for defining the maximum value of the analog output voltage, usually called full-scale, that, in this case, is equal to 7 V [3,11].

In Fig. 1.29, V_{AN} is the analog output.

Since the DAC has 3 input digital data, there are $8 (=2^3)$ possible binary combinations that it can transform into a corresponding analog output. Therefore, for a specific digital combination defined at its inputs, there will be a corresponding analog value at its analog output, which is proportional to the reference voltage. For example, ideally when the binary combination 000_2 is applied at E_2 , E_1 , and E_0 , there will be no voltage at its analog output, i.e., 0 V. Likewise, when the binary combination 001_2 is applied through the E_2 , E_1 , and E_0 inputs, the analog output (V_{AN}) will be equal to $V_{REF}/(2^{\text{number_of_inputs}} - 1)$ volts, which in this case will be $7\text{ V}/(2^3 - 1) = 7\text{ V}/7 = 1\text{ V}$, and so on. Consequently, for each digital value applied at its inputs, an analogue value corresponding to V_{REF} will be defined at its output and is given by Eq. (1.4) [3,11].

$$V_{AN} = \frac{(E_2 \cdot 2^3 + E_1 \cdot 2^2 + E_0 \cdot 2^0) \cdot V_{REF}}{2^{\text{number_of_inputs}} - 1} \quad (1.4)$$

Table 1.14 presents the possible binary combinations of the inputs of the DAC and the corresponding analog values at its output, regarding a V_{REF} of 7 V [3, 11].

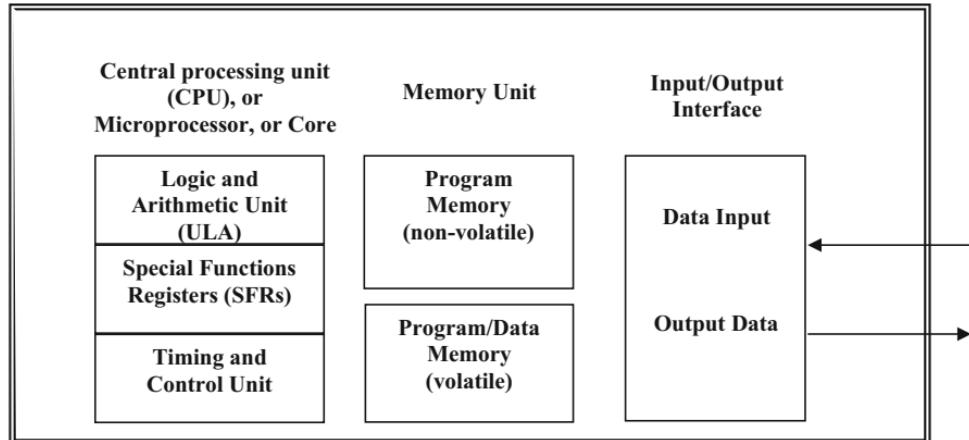


Fig. 1.30 Basic blocks of a microcomputer

DACs are used in a broad spectrum of integrated circuit applications, such as audio amplifiers, video encoder, display electronics, data acquisition systems, calibration, motor control, data distribution system, digital potentiometer, etc. [3, 11].

1.9 Microcomputer

Microcomputer is a compact electronic equipment, whose main purpose is to emulate or reproduce the “human system,” i.e., it is an electronic human. These machines, different from the humans, present high processing speed (high-frequency response), usually high information storage capacity (programs and data stored in memories), and high reliability because they are implemented with integrated circuits (ICs). Their basic structure consists of three basic blocks: central processing unit (also named CPU, microprocessor, or core), memory unit (program and data), and input and output data unit, according to Fig. 1.30 [2–11].

Before explaining the parts of the microcomputer, some other definitions are necessary, which will help us better understand the text.

1.9.1 Microprocessor Instructions

Each microprocessor has a processing capability that is defined by its manufacturer. This processing capacity is defined by the number of operations (instructions) that the microprocessor is able to perform. An instruction defines a single operation (command) that the microprocessor can execute at a time. These operations can be classified as data transfer from one place to another (special function registers, program and data memories, input/output ports, etc.), logical/arithmetic operations, and unconditional and conditional jumps, among others. An instruction can have the

size of one or more bytes, depending on its function. The larger its size, the longer it will take to be run [2–11].

1.9.2 Software

It is a set of microprocessor instructions properly organized by a programmer (a software expert), with the goal of informing/teaching the microprocessor step by step what the computer system must do over time. There are several program classifications, which are basic (operating systems: MacOS, Windows, etc.), utilities (Norton Utilities, Anti-virus, etc.), and applications (MS Word, etc.). The pieces of software to be run must always be stored in an information storage unit, i.e., in a memory (non-volatile or volatile). Usually, the basic pieces of software are stored in non-volatile memories, and the utilities and applications are stored in volatile memories [2–11].

1.9.3 Hardware

It is constituted by electronic parts which are used to build a microcomputer, such as the mother board, the video card, the network card, the controller of the hard disk (HD) of a microcomputer, etc. [2–11].

1.9.4 Firmware

It is a software that is exclusively stored in a non-volatile memory (ROM/PROM/EPROM/EEPROM/FLASH) of a microprocessor-based equipment. At a minimum, the firmware is responsible for programming the operation conditions of the hardware (input and output ports, serial communication, timers/counters, initialization of variables of the operating system, pieces of software of a simple calculator, car alarm, microwave oven, refrigerator, etc.) [2–11].

1.9.5 Central Processing Unit (CPU) or Microprocessor or Core

The central processing unit (CPU), also named microprocessor or core, is responsible for simulating/emulating the human brain. Physically, it is an integrated circuit (IC) constituted by millions/billions of transistors that integrate a variety of integrated circuits (ICs), such as registers, adders, comparators, timers, counters, transceivers, etc. The microprocessor is responsible for performing three main functions. The first one is to fetch the software in the memory of the computer system, instruction by instruction. This is done through read operations in the memory where the software is stored (non-volatile program memory or volatile program/data memory). It is important to highlight that the software of a computer

system is not allocated internally inside the microprocessor, but it resides in the memory unit of the computer system (Fig. 1.28). A piece of software is constituted by a set of instructions properly organized to meet a particular purpose, and an instruction may consist of one or more bytes. Once the reading operation of the instruction is performed, it is copied to an internal register, usually named instruction decoder register (IDR), if the instruction is 1 byte. This register is responsible for decoding it, i.e., it identifies the instruction function (move, logic, arithmetic, jumper instructions, etc.). After these two activities, named fetch cycle, a special function register called “program counter” (PC) is incremented by the number of bytes that define this instruction. If the instruction is defined by 1 byte, the PC is incremented by one; if the instruction is defined by 2 bytes, the PC is incremented by two; and so on. The PC function is responsible for defining the next address for the microprocessor to perform the fetch cycle (fetching and performing the decodification of a single instruction of a piece of software). After these activities, the microprocessor executes the so-called execution cycle, i.e., it runs the instruction fetched and decoded (perform its function). These two cycles (fetch and execution) performed by the microprocessor are responsible for running a piece of software instruction by instruction in a sequence designed by a programmer. This sequence is only broken if an instruction of the microprocessor is capable of altering the PC value (unconditional or conditional jumpers, for instance) [2–11].

The microprocessor consists of three (3) parts, which are formed by a logic and arithmetic unit, special function registers (SFRs), and a timing and control block [2–11].

I. Arithmetic and Logical Unit (ALU) The ALU is primarily responsible for executing the arithmetic and logical operations which the microprocessor is able to perform. In addition, it is responsible for defining the numerical condition of the result, e.g., if it is equal to zero or different from zero; if it is greater than, equal to, or smaller than another number; if it is positive or negative; if it is even or odd; etc. This is performed through some signaling bits, named “signaling flags” or simply “flags,” which can be defined as logic zero (reset) or logic one (set). These flags can be tested through specific instructions (usually the conditional instructions). These instructions are usually outlined in the software to define whether certain tasks should be performed or not. This means that the microprocessor-based systems are capable of taking decisions as humans do, i.e., the computer systems acquire intelligence. This intelligence feature of these machines is generated through a piece of software which is designed by programmers (humans) [2–11].

II. Special Function Registers (SFRs) The special function registers are constituted by flip-flops, generally D-type, which we may perform read and write operations. When the CPU is de-energized, its stored data are lost because they have volatile characteristics. Generally, they are able to store [2–11]:

- One byte, if they are implemented with registers of 8 bits
- A double byte, address, or word, if they are implemented with registers of 16 bits

- A double address, or double word, if they are implemented with registers of 32 bits
- Etc.

A microcomputer is called an “8-bit microcomputer” if it is capable of processing data of 8 bits at a time through the read/write operations in the SFRs and memory, arithmetic/logic operations, etc. Analogously to microcomputers of 16 and 32 bits, the more bits processed in parallel at a time by a microcomputer, the greater their processing capacity. The unit to measure the processing capacity of a microcomputer is the MIPS (millions of information per second). It should be noted that the microprocessor is not an information storage unit (memory) because it only has a few internal registers that are normally used for temporary storage of data. In other words, its function is not to store large amounts of data, in contrast to the memory function. Large amounts of information should be allocated in the memories that belong to the hardware of the microcomputer system [2–11].

III. Timing and Control Unit The timing and control unit is responsible for controlling the data flow from the microprocessor to the memory units and to the input and output units and vice versa. The control of the data flow is based on the management of the control signals overtime for the initialization operation of the microprocessor (RESET signal), read operation (READ signal), write operation (WRITE signal) in the SFRs and memories, arithmetic/logical operations, etc. [2–11].

1.9.6 Memory Unit

The memory unit is composed of two (2) parts: (I) non-volatile and (II) volatile [2–11].

I. Non-volatile Memory The non-volatile memory part is responsible for storing the program (total or only the initial part), which defines the firmware of the computer system. The firmware has the function of defining the operation conditions of the computer system. A firmware usually defines the configuration (programming mode) of the interfaces of the computer system (input/output ports, timers/counters, serial communication, etc.). This basic software (firmware) must be implemented by an electronic technician called a programmer, who is usually an expert about the hardware and instructions of the microprocessor used. This software must define the tasks, instruction by instruction, that the microcomputer must carry out over time. The firmware is never lost when the power supply is removed from the computer system due to the non-volatile characteristics of the memory. If a firmware is stored in the volatile memory and the power supply is removed from the computer system, this results in the functionality loss of this electronic equipment. The non-volatile memories normally used in computer systems are ROM, PROM/OTP, EPROM, EEPROM, and Flash. These memories are usually read-only, and consequently we

cannot perform write operations. Besides, the constants (fixed values) used for the process management are stored in non-volatile memories [2–11].

II. Volatile Memory The main function of the volatile memory is to store the data, i.e., the variables to be controlled through the program of the computer system, considering that all the program (software) is stored in the non-volatile memory (firmware). To illustrate, consider a computer system controlling a given activity (performing an automation). It is usually necessary to perform read operations of the variables to be controlled by the computer system. Generally, the control variables are defined by the external world through keys, sensors, keyboards, etc. These variables must be read through input interfaces and stored in the volatile memory. Later, they must be read and analyzed (processed) by the microprocessor, through a program (software), for the computer system to act in the environment that is being controlled. The RAM is usually used to store these control variables. Moreover, there are computer systems that allow that the programs be run in the RAM. Some examples of these computer systems are the notebooks, laptops, cellular phones, etc. In this type of computer systems, the firmware is mainly responsible for programming (configure) the hardware and subsequently to load an operational system in the RAM, aiming that other programs can be run [2–11].

1.9.7 Input/Output Units (I/O)

The data input/output units have the function of performing the communication (data exchange) between the environment to be controlled and the computer system, similarly to the human behavior with the environment [2–11].

I. Input Unit The input unit has the capability to obtain data (variables to be controlled) from the environment to the computer system. These variables can be defined through sensors, keys, switches, keyboards, hard disks, etc. [2–11].

II. Output Unit The output unit is responsible for providing the response of the computer system to the environment which we desire to control, e.g., an output interface with LEDs, which shows a binary value of operation conditions of a machine; a value of the temperature of a machine which is shown on the 7-segment/LCD display; the activation of a relay to turn on a DC motor; producing sounds through a speaker; etc. [2–11].

1.10 Microcontroller

In the last 30 years, microelectronics has been one of the areas of electrical engineering that has developed strongly. The “micro” at the beginning of the word microelectronics stands for the study of semiconductor devices (diodes, transistors, sensors, etc.) with micrometric dimensions ($10^{-6}/10^{-9}$ of the meter)

and how these devices can be used in the analog and digital CMOS IC applications. Another objective of this area is to further reduce the devices' dimensions more and more in order to increase the integration capacity and processing speed of these devices in CMOS ICs. This path was hard to be created and it took a long time, but today it has been reached in the nanoelectronics era [2–11].

In the past, a computer system was built with different CMOS ICs, each of them with a specific function, for instance, microprocessor, ROM, RAM, timers/counters, serial communication interface, etc. With the advance of microelectronics, today nanoelectronics can integrate many different CMOS ICs into a single IC. So, the process of integrating the basic blocks of a computer system (microprocessor, non-volatile and volatile memories, and different input/output interfaces) into a single CMOS IC is entitled "microcontroller" [2–11].

In the beginning, the microcontrollers were used in simple applications (car alarms, wrist watches, calculators, microwave ovens, etc.), in which a microprocessor with high processing capacity would not be suitable to perform this application, mainly because of the high cost. They were generally used in applications which did not need to manipulate and store high amounts of data, but only perform simple control tasks. They are usually very cheap, and their costs tend to be further reduced in the next years due to the large-scale production [2–11].

There are several manufacturers of microcontrollers, such as Texas Instruments, ATMEL, Microchip, etc. [2–11].

1.11 Flowchart

The flowchart is a graphical tool used to implement pieces of software. It illustrates how the objective of the software is achieved. In reality, it presents the strategy used by the software programmer to accomplish the objectives of the project. Besides, the flowchart shows the processing flow that the microprocessor performs in order to reach the objectives of the project [2–11].

To simplify, we are considering only four (4) basic graphic symbols to represent a complete flowchart: the connecting line, the ellipse, the rectangle, and the lozenge [2–11].

1.11.1 Connection Line

It represents the processing flow to be performed by the microprocessor. It is also used to connect the basic blocks of a flowchart [2–11].

1.11.2 Ellipse

It represents the beginning and the end of the software. When it is used at the beginning of the software, we must write its name in the ellipse. This name must be

related with its function. To illustrate, a routine that reads the keys of a keyboard could be named “read keyboard.” Other examples are: sum of 2 bytes, the highest value of the buffer, etc. This facilitates the understanding and documentation of the software. When the ellipse is used at the end of the flowchart, we should write “END and the software name” or simply “END.” If it is a subroutine, we must write an instruction such as Return or simply RET [2–11].

1.11.3 Rectangle

It represents the processing to be performed by the microprocessor, such as the move operations, arithmetic and logic operations, etc. [2–11].

1.11.4 Lozenge

Its function is to represent the execution of a test and the decision-making. The test can be performed to verify the condition of a bit, i.e., if it is equal to zero or one. Besides, it can verify if a value that was compared with others (subtract operation) is higher, smaller, different, or equal to it. After the test, the decision-making is done in two ways: (I) if the condition of the test is met, the microprocessor is able to jump to any other memory location that the programmer defined in the own instruction that is responsible for performing this test and decision-making. It is important to highlight that the jump to another memory location only happens if the test is achieved. (II) If the condition is not satisfied, the microprocessor will not perform the action described above, and consequently it will run the instruction subsequently to this instruction of test and decision-making (the next instruction) [2–11].

1.12 Sequential Programming

It is organized with instructions written sequentially, as illustrated in Fig. 1.31 [3, 11].

Initial address of the main program:

instruction 1

:

instruction n

Initial address of the loop of the main program:

instruction m

:

instruction p

unconditional jump to the **Initial address of the loop of the main program**

End

Fig. 1.31 Structure of a sequential program

This type of structure is characterized by its initial address and is usually ended by the word End. If the program must be run all day long, the last instruction must be an unconditional jump to the memory address in which the loop of the main program is defined [3, 11].

It is important to highlight that several processes can be performed simultaneously when we have a loop of the main program. This is possible because the processing speed of the microprocessors/microcontrollers (millions/billions of instructions per second) is much faster than the velocity of the processes of machines which the humans are used to manipulate (milliseconds, seconds, minutes, etc.). For instance, consider a machine that produces orange juice and presents several sub-processes to be performed: (I) put the oranges in the production line, (II) wash the oranges, (III) cut the oranges, (IV) squeeze the oranges, and (V) bottle the orange juice. Bear in mind that each process is implemented by a subroutine, and as the processing speed of the microprocessors/microcontrollers is much faster than the processes of the orange juice machine, when the subroutine that is responsible for processing I is run, the oranges are put in the production line. After running subroutine 1, the microprocessors/microcontrollers are also going to execute the subroutine that is responsible for washing the oranges, and therefore if there are oranges to be washed, they will be washed and so on. While this machine is turned on, all processes will be run. If this machine is working all day long, the orange juice will also be produced all day long [3, 11].

Figure 1.32 illustrates the typical flowchart of a sequential piece of software.

The flowchart of Fig. 1.32 is usually used to perform simple and non-repetitive processing activities. It always contains at least five basic blocks, which are [3, 11]:

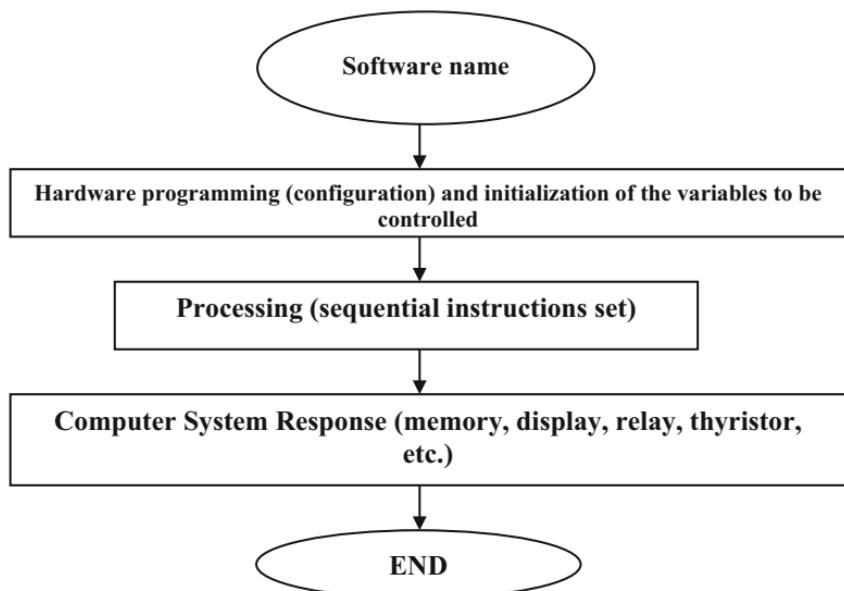


Fig. 1.32 Typical flowchart used for sequential pieces of software

1. The first block (ellipse) must contain the name of the program associated with its objective or its function.
2. The second block (rectangle) must contain the hardware programming (configuration), i.e., the way in which the hardware will operate (definition of the number of input/output ports that will be used to control a machine, quantity of external interrupters that will be used and their operation mode, Timer/Counter operation mode, serial communication mode, etc.). In addition, the initial values of the control variables must be defined too. This action has the function to define the starting point for the operation of the computer system.
3. The third block (rectangle) is responsible for describing the instructions that will be performed by the microprocessor (processing) based on the control variables in order to generate the output information from the computer system to the environment to be controlled (house, industry, car, microwave oven, rocket, etc.). This processing is performed through the move instructions, arithmetic/logic operations, etc.
4. The fourth block (rectangle) represents the instructions which are responsible for acting in the environment to be controlled as a response of the processing performed by the microprocessor, based on the control variables.
5. The last block (ellipse) symbolizes the end of the software/subroutine. If it is a subroutine, the programmer must define the instruction return (RET) in this ellipse.

1.13 Programming with Looping Structure

Figure 1.33 illustrates a classic flowchart to represent a programming with a looping structure [3, 11].

This flowchart of Fig. 1.33 is usually used in order to perform the same analyses with different data contained in a memory buffer (subsequent memory locations where there are a lot of data of the same type: age, weight, height, machines in operation, quantity of money of bank accounts, etc.), e.g., to calculate the quantity of individuals with age higher than or equal to 40 years old regarding a database of 100 individuals. Notice that the solution strategy to solve this software project is to compare the age of each individual with the value 40. If it is higher than or equal to 40, a counter must be incremented in one unit. Besides, this counter must be initialized in the beginning of the software (initial conditions) with its value equal to zero. Note that this same procedure must be performed regarding the 100 individuals (100 times) [3, 11].

This flowchart presents at least five (8) basic blocks, which are [3, 11]:

1. The first block (ellipse) should contain the name of the software. It must always be correlated with its purpose or its function.
2. A second block (rectangle) must program (configure) the hardware and define the control variables with their initial values. Generally, pieces of software with looping structure present a database to be analyzed. This database is allocated

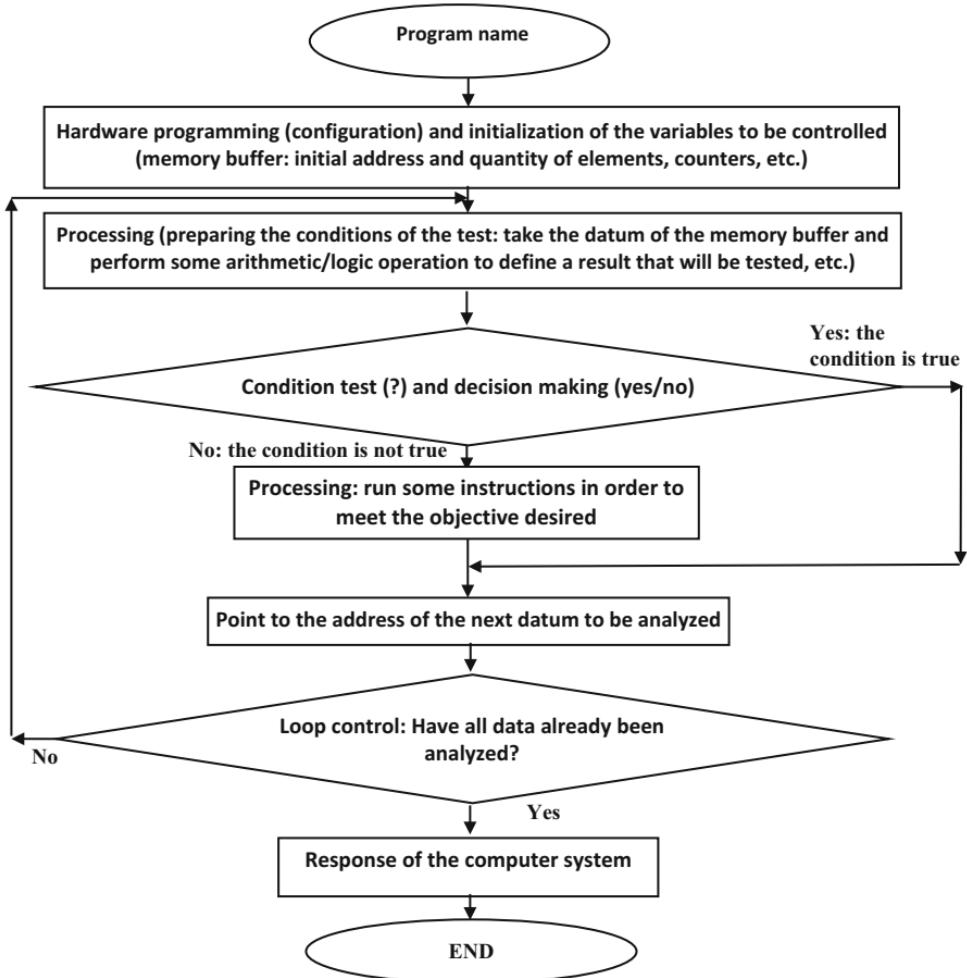


Fig. 1.33 Classic flowchart to represent a programming with a looping structure

- in memory where its elements occupy subsequent addresses of the data memory (memory buffer). Therefore, the control variables related to the memory buffer must be defined here in this block. A memory buffer can be defined in three different ways: through its initial and final addresses, through its initial address and the quantity of elements of the memory buffer, and through the amount of database elements and its final address. Other variables with their initial values must be defined in this block, as for instance, counters, memory locations, etc.
3. The third block (rectangle) must perform some instruction(s) (arithmetic, logic, rotation, etc.) taking into account each datum of the database aiming the execution of the desired analysis, such as comparison instructions, arithmetic/logic instructions, etc.
 4. A fourth block (lozenge) is responsible for performing the test and decision-making. Usually, the test consists of checking some bits, named flags, which are

changed by arithmetic, logic, and rotate instructions. These flags are able to define the mathematic conditions of the result obtained through these instructions that use the ULA of the microprocessor (if it is higher than or equal to, if it is smaller than, if it is equal to, if it is different to, etc.). Subsequently, the decision-making is performed, i.e., if the condition is met (it is true), the microprocessor is able to jump to a memory address defined by the programmer. If the condition is not met (it is false), the microprocessor will run the next instruction after it.

5. A fifth block (rectangle) must contain the instructions that must be run to obtain the desired function of the software. In these circumstances, if the test performed in the previous item is true, these instructions will not be executed because this block will be jumped, according to the flowchart of Fig. 1.31. However, if it is false, the instructions of this block will be run and be able to meet the desired objectives. For instance, if you want to calculate the quantity of odd numbers of a memory buffer, the programmer must test if the number is even. If it is really an even number, the instructions of this block will not be run, as indicated in the flowchart of Fig. 1.31. However, if the number is odd, the instructions of this block will be run, and therefore in this case, the programmer must increment a counter of one unit.
6. The sixth and seventh blocks (rectangle and lozenge) have the objective of positioning to the memory address of the next datum to be analyzed (rectangle) and verifying if all data of the memory buffer were analyzed (loop control). The verification if all data were analyzed is usually done in two (2) ways:
 - 6.1. If the database is controlled through its initial address and quantity of elements of the memory buffer, the variable that contains the quantity of elements of the database in the lozenge block must be decremented by one unit and compared if it is equal to zero. When this variable is different to zero, the processing flow will be changed to the beginning of the program (block 3 of Fig. 1.31), and the same processing will be performed concerning the next data of the database. However, if the variable which contains the quantity of elements of the database is equal to zero, this means that all data were analyzed and the objective of the software was met,
 - 6.2. If the memory buffer is controlled through its initial and final addresses, the variable that contains the quantity of elements of the database in the lozenge block must be compared with the final address incremented by one unit in order to verify if all data were analyzed. When the memory addresses of the database are different from the final address of the memory buffer incremented by one unit, the processing flow will be altered to the beginning of the program (block 3 of Fig. 1.31), and the same processing will be performed concerning the next data of the database. If it is equal to the final address incremented by one unit, this means that all data were analyzed and the objective of the software was achieved.
7. The eighth block (rectangle) must contain the response of the software of the computer system. In this case, some information obtained of the database must probably be presented in some output interface (display, LEDs, transmitted to another computer system, etc.).

8. In this last block (ellipse), RET or END must be written, depending on whether it is a subroutine or not, respectively.

1.14 Structured Programming

Structured programming is also called “modular programming,” which forces the programmers to have a logic and structured thought to automate a specific software project. This approach is based on using subroutines. These subroutines must be independent from each other and have their own objectives. Therefore, regarding a specific application, the programmer must divide its main objective into several sub-objectives (modules by using subroutines). Besides, these subroutines must be properly prearranged in a sequential way in order to meet their main objectives, according to Fig. 1.34 [3, 11].

The structured programming of Fig. 1.34 is divided into two regions. The first one consists of the different subroutines, which are defined by their initial addresses, and their last instruction is RET (return). The second region defines the main program, which includes the loop of the main program, which contains the different calls to the subroutines [3, 11].

```
; Region of the Subroutines  
Initial address of the subroutine 1:  
instruction 1  
:  
instruction n  
RET  
:  
Initial address of the subroutine m:  
instruction m  
:  
instruction p  
RET  
:  
; Region of the Main program  
Initial address of the main program:  
instruction q  
:  
instruction r  
Initial address of the loop of the main program:  
instruction s  
:  
call Initial address of the subroutine 1  
:  
call Initial address of the subroutine m  
:  
instruction t  
unconditional jump Initial address of the loop of the main program  
END
```

Fig. 1.34 Example of a structured programming

To run a structured program, we must initially execute the first instruction from the initial address of the main program. Therefore, the instruction q is run until it finds the first call instruction. When the first call initial address of subroutine1 is run, the processing is transferred to subroutine1, and instruction 1 is run until it finds the instruction RET. This instruction is responsible for returning the processing to the main program, the next instruction after the call initial address of subroutine1 is run, and so on [3, 11].

Several programming languages are designed with this approach, such as C, Java, etc. [3, 11].

Some advantages are achieved by using structured programs (subroutines) [3, 11]:

- The amount of memory (ROM, EPROM, and Flash) is reduced and consequently the cost of the hardware design.
- It presents a modularity feature, in which each subroutine has a specific function (objective). These subroutines can be kept in a library in order to be used in future projects. The modularity facilitates the implementation of the software and the new projects, simulation, electrical/physical test, emulation, maintenance, modification, etc.
- The program can be clearly understood, since each subroutine represents each subprocess that must be implemented to achieve the final objective of the project.

Figure 1.35 illustrates a typical flowchart of a complete structured program.

The typical flowchart of a structured piece of software is composed of 4 blocks. The first one is composed by the different subroutines that define the objective of the computer system. The second block must contain the hardware programming/configurations and the initialization of the control variables, etc. The third block is

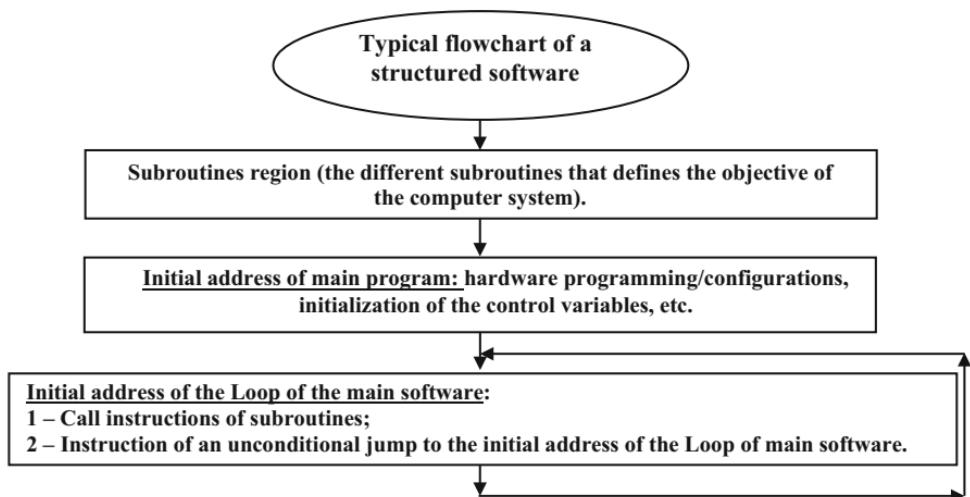


Fig. 1.35 Typical flowchart of a structured software

responsible for presenting the call instructions of the subroutines, and its last instruction must be an unconditional jump instruction to the initial address of the loop of the main program [3, 11].

1.15 Programming Languages

They are used by the programmers in order to design a software to be run on a particular computer system implemented with microprocessors or microcontrollers. They are classified into three (3) levels, which are low level, high level, and medium level [3, 11].

1.15.1 Low-Level Programming Language

It has the capacity of accessing, manipulating, and controlling the special function registers of the microprocessors/microcontrollers, the addresses and contents of memory locations, and the input and output devices of the hardware. This language is generally used by technically qualified users (electronic engineers and technicians). These programmers must be excellent experts regarding the hardware and instructions set of the microprocessor/microcontroller of the computer system. The ASSEMBLY programming language is an example of low-level language. Assembler is a piece of software which is responsible for transforming the Assembly language into the machine language (bytes). Each instruction/command of the microprocessor/microcontroller is represented through a byte or a set of bytes [11].

1.15.2 High-Level Programming Language

Its main feature is to unlink the hardware of computer systems from the software to be designed. The programmers do not need to know the hardware of the computer system. The focus of this programming language is to allow nontechnical programmers to develop their applications. The commands of these languages are very simple and easy to understand and manipulate. The BASIC and Java programming languages belong to this group of the programming language [11].

1.15.3 Medium-Level Programming Language

The medium-level language has as its main feature to meet the characteristics of both programming languages described above. Therefore, this programming language is very flexible for the programmers. The C programming language is an example of this category of language [11].

1.16 Basic Computer Architecture (Von Neumann Architecture)

The basic architecture of computer systems based on the von Neumann architecture is illustrated in Fig. 1.36.

This hardware architecture, proposed by John von Neumann, allows that software be stored and run in the same region where the data are stored. Besides, it consists of three buses (address, control signals, and data) shared with all basic blocks (CPU, memory, and input/output interfaces). These buses are responsible for performing the communication among the microprocessor (CPU) and the other blocks (memory: program and data) and input/output interfaces. The number of bits of each bus depends on the microprocessor/microcontroller used in the computer system [2–11].

1.16.1 Address Bus

This bus is unidirectional, and the microprocessor is responsible for defining the addresses of the program memory to fetch the instructions one by one sequentially (fetch cycle) for their execution (execution cycle). These two cycles define the instruction cycle. Besides, it is responsible for defining the memory addresses of programs, data, registers of the input/output interfaces, etc. Since there are some integrated circuits connected to the microprocessor/microcontroller, this bus can also be used to generate different selection signals, usually named chip enable (\overline{CS}) or chip enable (\overline{CE}), to enable only one integrated circuit at a time. These signals are generated through a decoder, whose inputs are connected to the address bus. Depending on the binary combination of the address bus (range of addresses for each integrated circuit), only an integrated circuit is enabled at a time in order to the microprocessor/microcontroller to be able to perform the communication with it. If a design error occurs, i.e., if two integrated circuits are enabled at the same time to exchange data with the microprocessor/microcontroller, as the data bus is common to both, two different data are defined in this bus at the same time. Consequently, if at

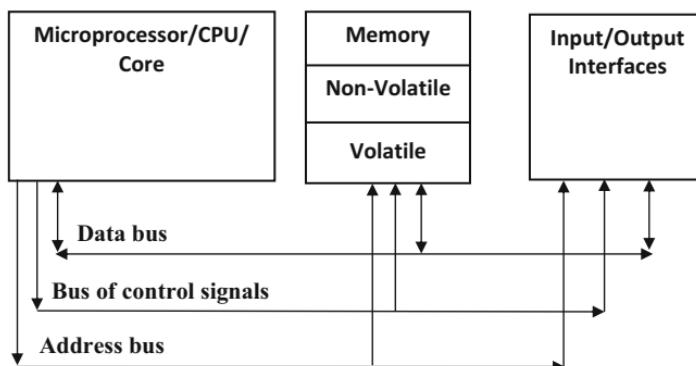


Fig. 1.36 The basic architecture of computer systems

least 1 bit of the data bus is defined with different logic levels, this generates a short circuit in this bit of this data bus, damaging these integrated circuits [2–11].

1.16.2 Timing and Control Bus

The microprocessor uses this bus to define the timing and control signals to manage the time and direction of the data flow regarding the read and write operations of the different devices (memory units or input and output units) connected to the microprocessor/microcontroller. This bus is unidirectional and is also defined by the microprocessor. Some examples of these signals are the read signal (\overline{RD}) and the write signal (\overline{WR}) [2–11].

1.16.3 Data Bus

The microprocessor uses this bus to receive data either from the memories or from the input/output devices through read operations, or to define the data to the memory or to the input/output devices through write operations. This bus is bidirectional. It is important to highlight that in some microprocessors or microcontrollers, the memory does not exchange data directly with the input/output devices and vice versa. This process occurs through a read operation in a device and after a write operation in the other device [2–11].

The sequence of events for the microprocessor to perform a read operation in memory or an input/output device is [2–11]:

- The CPU, through the timing and control block, sets the address of the memory or input/output device on the address bus for a certain time. Only one memory or input/output interface is selected and also only one memory location or register of the input/output interface.
- The CPU, through the timing and control block, sets the read control signal (\overline{RD}) on the timing and control bus for a certain time.
- The memory or input and output device provides the information (a byte, for example) for a certain time on the data bus, so that the microprocessor can obtain it (read operation). This byte is usually stored in one special function register of the microprocessor/microcontroller.

Likewise, the sequence of events for the microprocessor to perform a write operation in the memory or an input/output device is:

- The CPU, through the timing and control block, sets the address of the memory or input/output device on the address bus for a certain time. Only one memory or

input/output interface is selected and also only one memory location or register of the input/output interface.

- The CPU defines the datum (e.g., 1 byte) on the data bus.
- The CPU, via the timing and control block, sets the control signal (\overline{WR}) on the timing and control bus. Therefore, this datum is stored in the content of the memory location addressed by the address bus or in the content of the register of the input/output device.

1.17 How a Microcomputer Works

The microcomputer is an electronic machine capable of fetching and executing instructions defined in a piece of software that is stored in the memory. Each instruction has a function which the microprocessor must process to perform a given task (activity). After energizing a microcomputer, a system initialization signal is generated, usually called reset signal (see Sect. 1.6), by an external reset circuit required for the microprocessor of the computer system. This external circuit is connected to the reset pin of the microprocessor/microcontroller (any microprocessor or microcontroller must have this pin). One of the main functions of this signal is to perform the initialization of a special function register of a microprocessor (CPU) called program counter (PC). This register always contains the address of the next instruction to be fetched in program memory and executed by the microprocessor, whose initial value after the reset signal is predefined by the manufacturer, e.g., equal to 0000h for the MCS-51 family of microcontrollers from Atmel. In order for the computer systems to work suitably, at this address, it is necessary that the hardware designer has mapped a non-volatile memory (e.g., 8Kx8 EEPROM whose initial address is 0000h) and the programmer must record the software (firmware) of the computer system from this program memory address (0000h). Consequently, whenever the computer systems are powered-on, a reset signal will occur which will initialize the PC with the value 0000h. For instance, the microprocessor will perform the instruction cycle (constituted by the fetch cycle): fetch the instructions in the memory (perform the read instruction in the program memory and increment the PC in order for the address of the program memory address of the next instruction to be fetched and run) and execution cycle of an instruction, responsible for decoding (to find out its function) and executing it at the memory address defined by the PC. This process of the microprocessor regarding the fetch and execute instructions is cyclic and thus the software allocated in memory is run completely, instruction by instruction, in order for the computer systems to be able to reach their desired functional objective [2–11].

1.18 Solved Problems

1.18.1 Give Examples of Computer Systems

Solution

- Personal use: digital clocks, calculators, electronic calendars, cell phones, hand-held devices, etc.
- Home use: microwave ovens, washing machines, televisions, automatic door control systems, etc.
- Industrial use: microcomputers applied to manufacturing control, programmable logic controller (PLC), etc.
- Embedded electronics: car alarms, electronic injections, etc.

1.18.2 What Are the Advantages of Using a Computer System?

Solution Its high processing speed and reliability to perform tasks.

1.19 Proposed Problems

- 1.20.1 – What are the numeral systems mostly used by computer systems?
- 1.20.2 – Define a bit and an 8-bit register.
- 1.20.3 – What are the methods to convert a decimal number to binary?
- 1.20.4 – What is the procedure to perform a read operation of a 4×8 memory?
- 1.20.5 – What is the procedure to perform a write operation of a 4×8 memory of the byte 20h?
- 1.20.6 – What are the main types of non-volatile memory? Explain their functions.
- 1.20.7 – What are the main types of volatile memory? Explain their functions.
- 1.20.8 – Describe the function of the oscillator in a computer system.
- 1.20.9 – Describe the reset circuit of a computer system.
- 1.20.10 – How does an interface with a mechanical switch work?
- 1.20.12 – What is the procedure to decode/read the switches of a keyboard?
- 1.20.13 – How does an interface with LDR work?
- 1.20.14 – How does the analog-to-digital converter work?
- 1.20.15 – How does an interface with LEDs work?
- 1.20.16 – How does a 7-segment display work?
- 1.20.17 – How does an LCD work?
- 1.20.18 – What is a PWM?
- 1.20.19 – How can we manage the speed of DC motors?
- 1.20.20 – How can we manage the speed of AC motors?
- 1.20.21 – How does the digital-to-analog converter (DAC) work?
- 1.20.22 – Define microcomputer.
- 1.20.23 – Define microprocessor.

- 1.20.24 – Define microprocessor instructions and software (program).
- 1.20.25 – Define microcontroller.
- 1.20.26 – Explain the importance of using flowcharts to implement the software.
What are the geometric symbols used to describe it?
- 1.20.27 – Define the instructions of a microprocessor.
- 1.20.28 – Define the software of a microprocessor.
- 1.20.29 – Define the firmware of a microprocessor.
- 1.20.30 – What is a sequential programming? Give its equivalent flowchart.
- 1.20.31 – What is a programming with looping structure? Give its equivalent flowchart.
- 1.20.32 – What is structured programming? Give its equivalent flowchart.
- 1.20.33 – How is the programming language classified?
- 1.20.34 – Describe the basic computer architecture (von Neumann architecture).
- 1.20.35 – How do the computer systems work?

References

1. Tocci RJ, Widmer NS, Moss GL (2007) Digital systems principles and applications, 10th edn. Pearson, Prentice Hall
2. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
3. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
4. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
5. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
6. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
7. Atmel Corporation (1997) AT89 Series Hardware Description (0499B-B), Dec 1997
8. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
9. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D-MICRO), June 2008
10. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
11. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

8051 Core Microcontrollers

2.1 Introduction

This chapter describes the family of 8051 core microcontrollers (originally from Intel), considering the main physical (encapsulation and pinning), constructive, electrical (internal architecture), and operating features (Idle and low power, among others).

This microcontrollers' family is made by many manufacturers, such as Texas Instruments, Philips, Analog Devices, and Siemens, among others, mainly because its patent has already become public domain. It has an established design (reliable), and consequently it is commonly used by designers around the world.

The family of 8051 core microcontrollers is used in different applications of machine control, such as keyboards for personal computers, home alarms, implantable medical devices (IMD), aerospace, embedded applications, etc. These characteristics are the great motivation for the study of these microcontrollers.

2.2 General Features

The 8051 microcontroller is the original member of the MCS-51 family of Intel. Its main characteristics are [1–10]:

- 8-bit CPU optimized for machine and process control applications.
- Powerful Boolean processing capability, including individual bit logic.
- It is able to address up to 64 Kbytes of program memory.
- It is capable of addressing up to 64 Kbytes of data memory.
- It presents an internal program memory of 4 Kbytes.
- It contains an internal RAM memory of 128 bytes.
- It has 32 bidirectional input and output bits which can be addressed and programmed individually. They are arranged in four 8-bit registers called ports (P0, P1, P2, and P3).

- It presents two 16-bit programmable Timers/Counters.
- It has five programmable interrupt inputs, of which three are internal and two external. The two internal interrupts are related to the Timers/Counters (Timer 0 and Timer 1), and the other is related to the interruption of the serial communication interface (transmission and reception). The two external interruptions (Ex0 and Ex1) are usually used to manage external events derived from sensors.
- It presents an asynchronous serial communication channel.
- It has an internal oscillator.
- It presents a 16-bit address bus for accessing external devices (program and data memories, input/output interfaces, etc.). The P0 and P2 ports are used for this purpose.

2.3 Internal Architecture

The basic architecture of the 8051 microcontroller is illustrated in Fig. 2.1 [1–10].

Table 2.1 presents the members of the 8-bit MCS-51 family from Intel. The original member is the 8051 microcontroller, and the others are variations of it, which have incorporated new features, such as more quantity of internal memories, interruptions, Timers/Counters, ports, etc. [1–10].

The letter C located in the middle of the name 80C51BH (fourth line of Table 2.1) means that the devices were implemented with the CHMOS technology. They are fully compatible with the 8051 microcontroller and also absorb less electric current

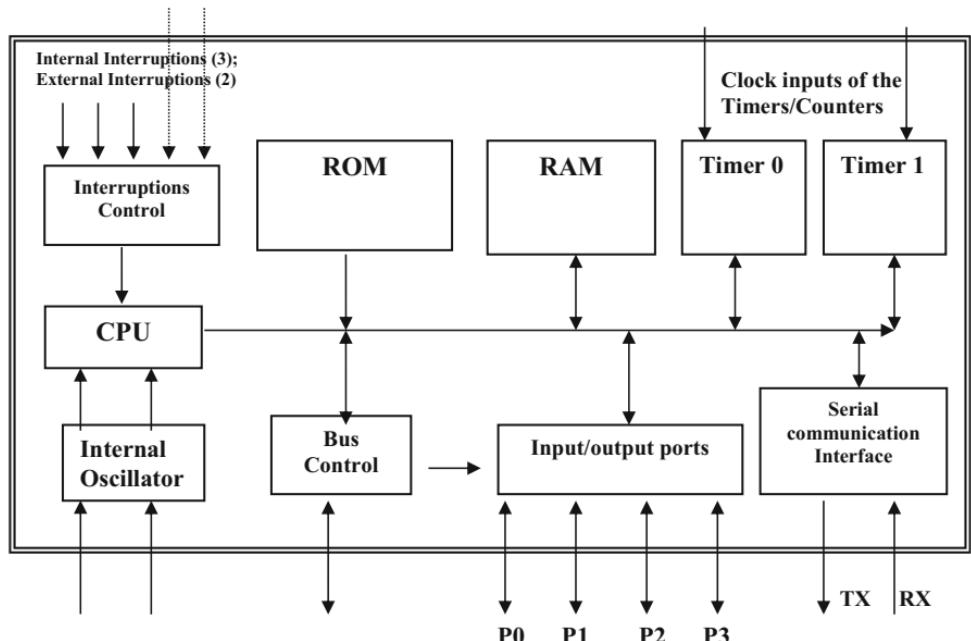


Fig. 2.1 The basic architecture of the 8051 microcontroller

Table 2.1 The Intel 8-bit family of microcontrollers

Device	ROMless	With EPROM	ROM quantity	RAM quantity	Input/output ports	Timers/Counters	UART	DMA	A/D inputs	Interruptions number	Low power and Idle modes
8051	8031	—	4 K	128	4	2	X			6/5	
8051AH	8031AH	8751AH 8751BH	4 K	128	4	2	X			6/5	
8052AH	8032AH	8752BH	8 K	256	4	3	X			8/6	
80C51BH	80C31BH	87C51	4 K	128	4	2	X			6/5	X
80C52	80C32	—	8 K	256	4	3	X			8/6	X
83C51FA	80C51FA	87C51FA	8 K	256	4	3	X			14/7	X
83C51FB	80C51FA	87C51FB	16 K	256	4	3	X			14/7	X
83C152JA	80C152JA	—	8 K	256	5	2	X	2		19/11	X
—	80C152JB	—	—	256	7	2	X	2		19/11	X
83C152JC	80C152JC	—	8 K	256	5	2	X	2		19/11	X
—	80C152JD	—	—	256	7	2	X	2		19/11	X
83C452	80C452	87C452P	8 K	256	5	2	X		9/8		X

than the one observed in the HMOS technology. They present two reduction modes of the power consumption. The first mode, called Idle, is programmed by software which turns off the CPU, while the RAM and other internal devices continue to operate normally. In this mode, the electric current of the CPU is reduced by 15%. The second mode is simply called low power, also programmed by software. Most internal devices are turned off, and only the internal RAM continues to operate. In this mode, the consumption of electric current is less than 10 µA [1–10].

The letters BH of the name 80C51BH use a more sophisticated technology than the A described before, and they are functionally compatible with the HMOS technology. They are fully compatible with the HMOS and CHMOS technologies [1–10].

2.4 Pin Description of the 8051 Microcontroller

The 8051 microcontroller is a 40-pin integrated circuit (IC) according to Fig. 2.2 [1–10].

Additional details regarding the pins of the 8051 microcontroller are indicated in Table 2.2 [1–10].

2.5 Memory Organization

Figure 2.3 illustrates the memory organization of the MCS-51 family of microcontrollers from Intel [1–10].

2.5.1 The Program Memory (Internal and External)

The main functions of the program memory are [1–10]:

- To store the firmware (program that gives the basic functionality of computer systems)
- To store constant (fixed) values that are used during the processing of a given task. To illustrate the value of the number (3.141592 ...) to be used in calculations that are necessary for processing a process and conversion tables, among others.

The data memory has the main function of storing the variables to be controlled, such as the read value of a temperature sensor that varies over time and an auxiliary variable used during certain calculations, among others. In addition, programs can also be stored (applicative software) in this type of memory, such as text editors (MS Word) of a personal computer, etc. [1–10].

The program and data memories of the MCS-51 microcontrollers' family from Intel are logically separated. The logical separation allows the data memory to be accessed by 8-bit direct addressing and by 16-bit indirect addressing, controlled by

Reset input: defined by external circuit or back-up supply voltage to the internal RAM	9	Reset/V _{P0}	\overline{EA}	External Access: input control signal programmed by hardware in order to define the access to the external (V_{ss}) or internal (V_{cc}) program memory
Clock 2 input: defined by an external oscillator	18	X2	ALE/PROG	Address Latch Enable: Output control signal used to demultiplex (capture) the least significant byte of the address/data bus ($AD_{7:0}$; 39-32 pins of 8051) or program pulse input during the programming of the internal program memory
Clock 1 input: defined by an external oscillator	19	X1	\overline{PSEN}/V_{PP}	Program Store Enable: output control signal to read external program memory or programming voltage (V_{PP})
Bit 0 of port 0 (P0.0) or bit 0 of the least significant byte of the address bus which is multiplexed with the bit 0 of the data bus (AD ₀)	39	P0.0/AD0	P2.0/A8	Bit 0 of port 2 (P2.0) or bit 8 of the most significant byte of the address bus
Bit 1 of port 0 (P0.1) or bit 1 of the least significant byte of the address bus which is multiplexed with the bit 1 of the data bus (AD ₁)	38	P0.1/AD1	8	Bit 1 of port 2 (P2.1) or bit 9 of the most significant byte of the address bus
Bit 2 of port 0 (P0.2) or bit 2 of the least significant byte of the address bus which is multiplexed with the bit 2 of the data bus (AD ₂)	37	P0.2/AD2	P2.2/A10	Bit 2 of port 2 (P2.2) or bit 10 of the most significant byte of the address bus
Bit 3 of port 0 (P0.3) or bit 3 of the least significant byte of the address bus which is multiplexed with the bit 3 of the data bus (AD ₃)	36	P0.3/AD3	P2.3/A11	Bit 3 of port 2 (P2.3) or bit 11 of the most significant byte of the address bus
Bit 4 of port 0 (P0.4) or bit 4 of the least significant byte of the address bus which is multiplexed to the bit 4 of the data bus (AD ₄)	35	P0.4/AD4	P2.4/A12	Bit 4 of port 2 (P2.4) or bit 12 of the most significant byte of the address bus
Bit 5 of port 0 (P0.5) or bit 5 of the least significant byte of the address bus which is multiplexed with the bit 5 of the data bus (AD ₅)	34	P0.5/AD5	0	Bit 5 of port 2 (P2.5) or bit 13 of the most significant byte of the address bus
Bit 6 of port 0 (P0.6) or bit 6 of the least significant byte of the address bus which is multiplexed with the bit 6 of the data bus (AD ₆)	33	P0.6/AD6	P2.6/A14	Bit 6 of port 2 (P2.6) or bit 14 of the most significant byte of the address bus
Bit 7 of port 0 (P0.7) or bit 7 of the least significant byte of the address bus which is multiplexed with the bit 7 of the data bus (AD ₇)	32	P0.7/AD7	P2.7/A15	Bit 7 of port 2 (P2.7) or bit 15 of the most significant byte of the address bus
Bit 0 of port 1 (P1.0)	1	P1.0	5	P3.0/RXD
Bit 1 of port 1 (P1.1)	2	P1.1		P3.1/TXD
Bit 2 of port 1 (P1.2)	3	P1.2		P3.2/ $\overline{INT0}$
Bit 3 of port 1 (P1.3)	4	P1.3	1	P3.3/ $\overline{INT1}$
Bit 4 of port 1 (P1.4)	5	P1.4		P3.4/T0
Bit 5 of port 1 (P1.5)	6	P1.5		P3.5/T1
Bit 6 of port 1 (P1.6)	7	P1.6		P3.6/W _R
Bit 7 of port 1 (P1.7)	8	P1.7		P3.7/R _D
				Bit 0 of port 3 (P3.0) or serial data input
				Bit 1 of port 3 (P3.1) or serial data output
				Bit 2 of port 3 (P3.2) or external interrupt 0
				Bit 3 of port 3 (P3.3) or external interrupt 1
				Bit 4 of port 3 (P3.4) or external clock input of the Timer/Counter 0
				Bit 5 of port 3 (P3.5) or external clock input of the Timer/Counter 1
				Bit 6 of port 3 (P3.6) or write control signal to the external data memory
				Bit 7 of port 3 (P3.7) or read control signal to the external data memory

Fig. 2.2 Pinning the 8051 microcontroller

the special function register called data pointer (DPTR), which will be presented later [1–10].

Up to 64 Kbytes of program memory can be accessed. When the control signal named external access (\overline{EA}) is equal to logic 1, the microcontroller versions

Table 2.2 Additional details about the 8051 microcontroller pins

Name	Symbol	Pin number	Description/function
Supply voltage	V_{CC}	40	High-energy level of the supply voltage
Ground	V_{SS}	20	Low-energy level of the supply voltage
Port 0 or the least significant byte of the address bus which is multiplexed with the data bus (AD_{7-0})	P0: P0.0 a P0.7 or AD_{7-0} : AD0-AD7	39–32	Port 0 is an 8-bit bidirectional input or output in open drain. Operating as an output port, each pin can absorb 8 low-power Schottky transistor-transistor (LS TTL) inputs. By writing logic 1 on the pins, they float, and these pins can be used as high impedance inputs. P0 is also the least significant address bus (AD_{7-0}) that is multiplexed with the data bus during the access to an external program and/or data memories. In this application, internal pull-ups are used when writing logic 1 and can supply or absorb up to 8 LS TTL inputs. Port 0 also receives the code bytes during the EPROM programming, and they exit with the bytes of the codes during the verification of the program that was recorded in the ROM and EPROM. External pull-ups are required during program verification
Port 1 or T2, T2EX (external clock input to the Timer/Counter 2, only for the 8032/52), P1.2 a P1.7	P1: P1.0 a P1.7, or T2 (P1.0) and T2EX (P1.1), only for the 8032/52	1–8	Port 1 is an 8-bit bidirectional input/output (I/O) port with internal pull-ups. Their output buffers can supply or absorb 4 LS TTL inputs. By writing logic 1 on the pins, their logical levels can be defined through their internal pull-ups which must be connected with a specific input interface and therefore they can be used as inputs. When this port is programmed as an output (logic 0), its bits are externally taken to logic

(continued)

Table 2.2 (continued)

Name	Symbol	Pin number	Description/function
Port 2 or the most significant byte of the address bus (A_{15-8}) of the external program/data memories	P2: P2.0 a P2.7 or A8 – A15	21–28	0, and consequently they can provide a current through their internal pull-ups. In the 8032AH/52AH, pins P1.0 and P1.1 also serve the functions of T2 (external clock input of the Timer/Counter 2) and T2EX (Timer/Counter 2 capture/reload trigger) Port 2 is an 8-bit bidirectional I/O port with internal pull-ups. Output buffers can supply or absorb 4 LS TTL inputs. By writing logic 1 on these pins, their logical levels can be defined through their internal pull-ups which must be connected with a specific input interface and therefore they can be used as inputs. When port 2 is programmed as an output (logic 0), its bits are externally taken to logic 0, and therefore they can provide electric current due to their internal pull-ups. Port 2 also is able to define the most significant address byte (A_{8-0}) during the programming of the external non-volatile memory and during access to external data memory using 16-bit addressing (MOVX @DPTR). In this application, it uses internal pull-ups when it defines logic 1. While accessing external data memory, it uses 8-bit addressing (MOVX @Ri). Port 2 defines the contents of the special function register (P2) and also receives the byte of the most significant address during non-volatile programming and verification

(continued)

Table 2.2 (continued)

Name	Symbol	Pin number	Description/function
Port 3 or serial data reception and transmission, external interruptions 1 and 0, external clock inputs of timers 0 and 1, write and read control signals from external RAM memories	P3.0 a P3.7 ou RXD, TXD, <u>INT0</u> , <u>INT1</u> , T0, T1, <u>WR</u> , <u>RD</u>	10–17	Port 3 is an 8-bit bidirectional I/O port with internal pull-ups. Its output buffers can supply or absorb 4 LS TTL inputs. By writing logic 1 on these pins, their logical levels can be defined through their internal pull-ups which must be connected with a specific input interface and therefore they can be used as inputs. When port 3 is programmed as an output (logic 0), its outputs provide electric current due to their internal pull-ups Port 3 also presents multiple functions: serial data reception and transmission, interruption inputs, clock inputs of the two Timers/Counters, and write and read control signals from external memories
Reset/V _{PD}	RST	9	Reset signal input. A high logic level on this pin of two machine cycles initializes the microcontroller (it defines the initial conditions in some special function registers, with the aim of starting the microcontroller, i.e., to run the firmware from its first instruction) V _{PD} : The backup supply voltage to the internal RAM if the V _{CC} drops below its specification (minimum value)
Address latch enable (ALE) or PROG	ALE/ <u>PROG</u>	30	Address latch enable (ALE): It is an output pulse to enable an external device (usually a latch/flip-flop) to obtain the least significant byte of the address/data bus, which is multiplexed with the data bus. In a normal operation, the ALE control signal is generated at a constant ratio

(continued)

Table 2.2 (continued)

Name	Symbol	Pin number	Description/function
			of 1/6 of the oscillator frequency and can also be used as a clock signal or an external timer. Besides, the ALE control pulse is activated at each access to the external data memory <u>PROG</u> : This input pin is also used to program the internal non-volatile memory
Program store enable	PSEN	29	Program store enable: It is the read control signal for the external program memory. When the device is executing the instruction cycle in the external program memory, it is activated twice in each machine cycle, except when the external data memory is accessed
External access enable or programming supply voltage	\overline{EA}/V_{PP}	31	External access enable: It must be connected to the V_{SS} to enable the microprocessor of the 8051 microcontroller to fetch and execute the instructions in the external program memory from the address 0000_h to $FFFF_h$. It must be connected to the V_{CC} to execute the software (firmware) contained in the internal non-volatile memory. However, if the security bit in the non-volatile is enabled, the device does not fetch the instructions in the external program memory V_{PP} : This pin must be defined with a 21 V programming power supply (V_{PP}) during the EPROM programming. This V_{PP} value depends on the CMOS ICs technology node used, and therefore the manufacturer is responsible for defining it
Input 1 of the external clock oscillator	XTAL1	19	Input 1 of the external clock oscillator
Input 2 of the external clock oscillator	XTAL2	18	Input 2 of the external clock oscillator

Memory Program

Data Memory

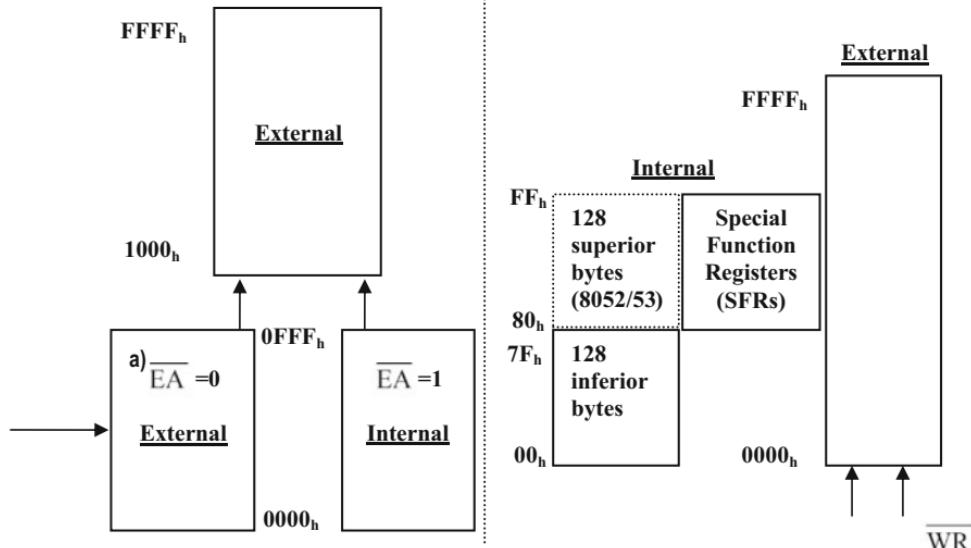


Fig. 2.3 Memory organization of the MCS-51 microcontrollers' family from Intel

incorporating ROM, EPROM, and Flash, the first 4 K (e.g., 8051AH), 8 K (e.g., 8052AH), and 16 Kbytes (e.g., 83C51FB) are, respectively, provided internally by the microcontroller. The remainder, if necessary, can be added externally. In this case, the firmware must be written in the internal program memory, since the CPU performs the instruction cycle (fetch and execution cycles) in this internal program memory [1–10].

If the **EA** is equal to logic 0, the microcontroller CPU performs the instruction cycle (fetch and execution cycles) in the external program memory. In the version without ROM, EPROM, or Flash, all the program memory must be stored externally [1–10].

If a microcontroller presents an internal program memory, but the **EA** is equal to logic 0, its internal program memory cannot be used to store the firmware because the execution cycle will be performed regarding the external program memory [1–10].

The program store enable (**PSEN**) control signal corresponds to the read control signal of the external program memory, and it is also used to perform the fetch cycle of the program instruction and to perform the read operation of data in the external program memory [1–10].

The initial address of the internal program memory must always be equal to **0000_h**, and its final address value can vary from **0FFF_h**, **1FFF_h**, or **3FFF_h**, depending on the microcontroller, i.e., 4 Kbytes, 8 Kbytes, or 16 Kbytes, respectively. However, it can be extended to the **FFFF_h** if we add external program memory [1–10].

Figure 2.4 illustrates the address map of the internal program memory of an 8051 microcontroller [1–10].



Fig. 2.4 Address map of the program memory of the MCS-51 microcontroller family

The 8051 microcontroller has five interrupt sources: external 0, Timer/Counter 0, external 1, Timer/Counter 1, and serial communication interface. For the 8052/8053, there is an additional interrupt: Timer/Counter 2 [1–10].

For each interruption, the manufacturer reserves only an 8-byte program memory space to allocate the service subroutines for these interruptions. The subroutines for the servicing of external 0, Timer/Counter 0, external 1, Timer/Counter 1, serial communication, and Timer/Counter 2 must be allocated in the following program memory address ranges from 0003h to 000Ah, from 000Bh to 0012h, from 0013h to 001Ah, from 001Bh to 0022h, 0023h to 002Ah, and from 002Bh to 0032h, respectively [1–10].

Generally, the space reserved for a service subroutine for an interrupt source in the program memory is of 8 bytes. If the software designer needs more program memory space to write this service subroutine, it is recommended that in the first address of this memory location, the designer writes an unconditional jump instruction to another program memory address available in the program memory, in which it can be completely allocated. Therefore, when an interruption occurs, the service subroutine will be executed from the initial address defined by the manufacturer. As in this first address, an unconditional jump instruction was written, it is run, and consequently the microprocessor transfers the processing flux to the address indicated by this unconditional jump instruction, in which the service subroutine was stored [1–10].

It is important to highlight that when a reset signal happens, the program counter (PC) is reset (it returns to 0000_h), and consequently the microprocessor performs the instruction cycle (fetch and execution cycles) in the program memory addressed by the contents of the PC, which in this case is equal to 0000_h. Thus, the software designer must store its software (firmware) from the address 0000_h. If the interruptions are used by this project, an unconditional jump instruction to another address available must be written in the program memory address from 0000h to 0002h (the unconditional jump usually uses 3 bytes to be stored in the program memory) in order for the microprocessor to be able to jump to the program memory addresses reserved for the service subroutines of the different interruptions. Figure 2.5 illustrates the strategy that must be used to store a piece of software (firmware) in the program memory, when the interruptions of the 8051 microcontrollers are used in a project [1–10].

The external program memory is accessed (read operation) by using the MOVC instruction, in which it is able to activate its read control signal (\overline{PSEN}) [1–10].

2.5.2 The Data Memory

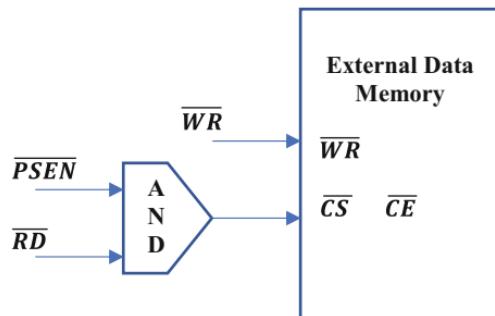
In this section, the features of the internal and external data memory are described.

I: The External Data Memory Regarding the external data memory as illustrated in Fig. 2.3, we can see that it presents the same addressing as the program memory (initial address, 0000_h; final address, FFFF_h). However, it can only be accessed by

Fig. 2.5 Strategy that must be used to store the software (firmware) in the program memory when interrupts are used

0034 _h	:
0033 _h	The software designer must store its software (firmware) from this address
0032 _h	:
:	:
0003 _h	External interruption 0 ($\overline{INT0}$)
0002 _h	
0001 _h	unconditional jump to 0033h, for instance
0000 _h	

Fig. 2.6 Hardware to run the software in external data memory



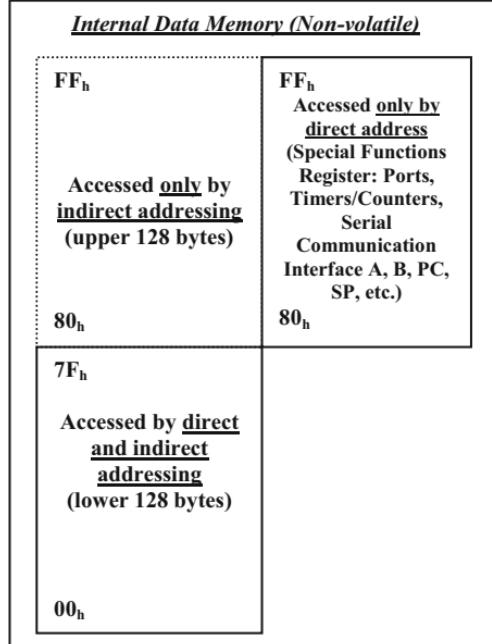
the MOVX instruction, in which it is capable of activating the read and write control signals (\overline{RD} and \overline{WR}).

It is important to highlight that not only can the software designer execute the software in the external program memory, but he can also run the software in the external data memory. This can be performed by combining the control signals \overline{PSEN} and \overline{RD} through a logic AND gate, whose output must be interconnected to the select chip (\overline{CS}) or enable chip (\overline{CE}) of the external non-volatile memory (data memory), according to Fig. 2.6 [1–10].

II: The Internal Data Memory The internal data memory is divided into three blocks: the lower 128 bytes, the upper 128 bytes (8052/53), and the special function registers (or simply SFRs), according to Fig. 2.7 [1–10].

The internal data memory is always addressed with 8 bits wide, i.e., we can only address 256 bytes (initial address, 00_h; final address, FF_h): the lower 128 bytes (initial address, 00_h; final address, 7F_h), the upper 128 bytes, only for 8052/53 members (initial address, 80_h; final address, FF_h), and the special functions registers or SFRs (initial address, 80_h; final address, FF_h). The lower 128 bytes can be accessed by using direct and indirect addressing. The upper 128 bytes can only be accessed by using indirect addressing. Additionally, SFRs can only be accessed by

Fig. 2.7 The addressing modes of the internal RAM and special function registers (SFRs)



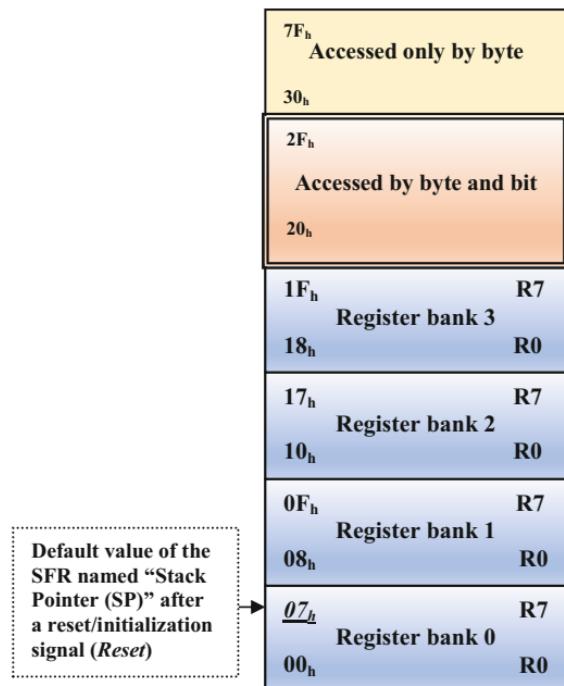
using direct addressing. It is important to highlight that the upper internal data memory and SFRs present the same addresses, but their accesses are performed through different types of addressing, as previously described. The mnemonics of the MSC-51 instructions of the movement to perform the direct and indirect addressing are, respectively, “MOV *direct*” and “MOV @*Ri*”. In this case, *direct* means the address of a memory location, and @*Ri* means an address defined by the content of the internal register *Ri*, which can only be the registers 0 (R0) and 1 (R1) of different register banks of the MSC-51 family. The software designer must not use the rest of the internal registers (R2, R3, R4, R5, R6, and R7) to perform the indirect addressing. It is not allowed to use them [1–10].

The lower 128 bytes of the internal RAM (00_h–7F_h) are present in all MCS-51 microcontrollers’ family, and they are divided into three parts, as illustrated in Fig. 2.8.

The first part consists of four banks of 8-bit registers. The addressing range used by these registers’ banks is given by the addresses from 00_h to 1F_h (initial 32 bytes). The second part consists of 16 bytes, whose initial and final addresses are 20h and 2Fh, respectively. This area of memory can be accessed by bit or byte. The access is subject to the type of addressing that will be used, depending on whether the instructions’ arguments are a bit or a byte. The initial and the final addresses of the third part of this internal data memory are 30_h to 7F_h [1–10].

II.A: Range of the Data Memory of the Registers’ Banks The initial 32 bytes of the internal RAM are grouped into four banks (register bank 0, 1, 2, and 3), each consisting of 8-bit registers, called R0, R1, R2, R3, R4, R5, R6, and R7. It is

Fig. 2.8 Structure of the lower 128 bytes of the 8051 microcontroller's internal RAM



important to highlight that these banks of registers can only be accessed one at a time. The selection of registers' banks is made through 2 bits, called “RS1” and “RS0” (register select 0 and 1, respectively). They are in bits 3 and 4 of the special function register named Program Status Word (PSW), referenced as PSW.3 and PSW.4. Depending on the RS1 and RS2 values, a single register bank is available for use, as illustrated in Fig. 2.9 [1–10].

The addressing mode by register performed by the microprocessor of the microcontroller uses these registers to perform some instructions. This addressing mode is capable of increasing the speed of the processing in relation to the use of other modes because they are run with a smaller quantity of clock cycles [1–10].

II.B: Range of the Data Memory Addressed by Byte and Bit The next 16 bytes of the internal data memory located above the register banks, whose memory addresses range from 20_h to 2F_h, constitute a block of memories addressed by both, byte and bit. The addresses of these bits range from address 00_h or 20h.0 (content of the bit 0 of the address 20h of the internal data memory) to 7F_h or 2Fh.7 (content of the bit 7 of the address 2Fh of the internal data memory), i.e., there are 128 addresses of bits in this range of memory, according to Fig. 2.10 [1–10].

The 8-bit contents of the internal data memory are addressed from 20_h to 2F_h (16 bytes), and the 128 contents of each bit of this range of the internal data memory are addressed from 00_h or 20h.0 to 7F_h or 2Fh.7 [1–10].

Fig. 2.9 Internal structure of the initial 32 bytes of internal RAM

Bit	7	6	5	4	3	2	1	0
(PSW) =	C	AC	F0	RS1	RS0	OV	-	P

RS1	RS0	Bank Selected	Registers selected	Addresses of the registers' banks
0	0	0	R0 a R7	00h a 07h
0	1	1	R0 a R7	08h a 0Fh
1	0	2	R0 a R7	10h a 17h
1	1	3	R0 a R7	18h a 1Fh

RS1	RS0	Address	Register	Bank
1	1	1Fh	R7	3
1	1	1Eh	R6	3
1	1	1Dh	R5	3
1	1	1Ch	R4	3
1	1	1Bh	R3	3
1	1	1Ah	R2	3
1	1	19h	R1	3
1	1	18h	R0	3
1	0	17h	R7	2
1	0	16h	R6	2
1	0	15h	R5	2
1	0	14h	R4	2
1	0	13h	R3	2
1	0	12h	R2	2
1	0	11h	R1	2
1	0	10h	R0	2
0	1	0Fh	R7	1
0	1	0Eh	R6	1
0	1	0Dh	R5	1
0	1	0Ch	R4	1
0	1	0Bh	R3	1
0	1	0Ah	R2	1
0	1	09h	R1	1
0	1	08h	R0	1
0	0	07h	R7	0
0	0	06h	R6	0
0	0	05h	R5	0
0	0	04h	R4	0
0	0	03h	R3	0
0	0	02h	R2	0
0	0	01h	R1	0
0	0	00h	R0	0

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
16 addresses of the contents of the Internal Data Memory (addressing byte)	128 addresses of the contents of bits of the Internal Data Memory (addressing by bit)							
2F _h	7F _h 2Fh.7	7E _h 2Fh.6	7D _h 2Fh.5	7C _h 2Fh.4	7B _h 2Fh.3	7A _h 2Fh.2	79 _h 2Fh.1	78 _h 2Fh.0
2E _h	77 _h 2Eh.7	76 _h 2Eh.6	75 _h 2Eh.5	74 _h 2Eh.4	73 _h 2Eh.3	72 _h 2Eh.2	71 _h 2Eh.1	70 _h 2Eh.0
2D _h	6F _h 2Dh.7	6E _h 2Dh.6	6D _h 2Dh.5	6C _h 2Dh.4	6B _h 2Dh.3	6A _h 2Dh.2	69 _h 2Dh.1	68 _h 2Dh.0
2C _h	67 _h 2Ch.7	66 _h 2Ch.6	65 _h 2Ch.5	64 _h 2Ch.4	63 _h 2Ch.3	62 _h 2Ch.2	61 _h 2Ch.1	60 _h 2Ch.0
2B _h	5F _h 2Bh.7	5E _h 2Bh.6	5D _h 2Bh.5	5C _h 2Bh.4	5B _h 2Bh.3	5A _h 2Bh.2	59 _h 2Bh.1	58 _h 2Bh.0
2A _h	57 _h 2Ah.7	56 _h 2Ah.6	55 _h 2Ah.5	54 _h 2Ah.4	53 _h 2Ah.3	52 _h 2Ah.2	51 _h 2Ah.1	50 _h 2Ah.0
29 _h	4F _h 29h.7	4E _h 29h.6	4D _h 29h.5	4C _h 29h.4	4B _h 29h.3	4A _h 29h.2	49 _h 29h.1	48 _h 29h.0
28 _h	47 _h 28h.7	46 _h 28h.6	45 _h 28h.5	44 _h 28h.4	43 _h 28h.3	42 _h 28h.2	41 _h 28h.1	40 _h 28h.0
27 _h	3F _h 27h.7	3E _h 27h.6	3D _h 27h.5	3C _h 27h.4	3B _h 27h.3	3A _h 27h.2	39 _h 27h.1	38 _h 27h.0
26 _h	37 _h 26h.7	36 _h 26h.6	35 _h 26h.5	34 _h 26h.4	33 _h 26h.3	32 _h 26h.2	31 _h 26h.1	30 _h 26h.0
25 _h	2F _h 25h.7	2E _h 25h.6	2D _h 25h.5	2C _h 25h.4	2B _h 25h.3	2A _h 25h.2	29 _h 25h.1	28 _h 25h.0
24 _h	27 _h 24h.7	26 _h 24h.6	25 _h 24h.5	24 _h 24h.4	23 _h 24h.3	22 _h 24h.2	21 _h 24h.1	20 _h 24h.0
23 _h	1F _h 23h.7	1E _h 23h.6	1D _h 23h.5	1C _h 23h.4	1B _h 23h.3	1A _h 23h.2	19 _h 23h.1	18 _h 23h.0
22 _h	17 _h 22h.7	16 _h 22h.6	15 _h 22h.5	14 _h 22h.4	13 _h 22h.3	12 _h 22h.2	11 _h 22h.1	10 _h 22h.0
21 _h	0F _h 21h.7	0E _h 21h.6	0D _h 21h.5	0C _h 21h.4	0B _h 21h.3	0A _h 21h.2	09 _h 21h.1	08 _h 21h.0
20 _h	07 _h 20h.7	06 _h 20h.6	05 _h 20h.5	04 _h 20h.4	03 _h 20h.3	02 _h 20h.2	01 _h 20h.1	00 _h 20h.0

Fig. 2.10 The block of the internal data memory addressed by byte and bit

Therefore, the procedure to transform a bit address represented by a byte (xy_h) into another which is defined by its memory address and the position of the bit (2rh. q, where q and r are, respectively, named quotient and rest) is [1–10]:

- Since there are 16 memory addresses in which their addresses are defined from 20h to 2Fh and each memory location has 8 bits, the data memory address represented in hexadecimal (xy_h ; 00_h - 7F_h) must be transformed into decimal and then divided by 8, obtaining the quotient (q) and the rest (r). The q must be transformed into hexadecimal and then added to 20h (initial address of this range of the internal data memory), and finally we must represent it as 2qh.r, according to Eq. (2.1) [1–10].

$$xy_h = (20h + qh).r = 2qh.r \quad (2.1)$$

For instance, consider the address of bit equal to 5Dh. Its decimal value is equal to 93₁₀ ($5 \cdot 16^1 + 13 \cdot 16^0$). Dividing 93₁₀ by 8₁₀, we have q and r equal to 11₁₀ and 5₁₀, respectively. Transforming 11₁₀ into hexadecimal, we have B_h. Therefore, using

Eq. (2.1), we have $(20h + Bh)h.5 = 2Bh.5$. Hence, the two representations of the bit address are $5D_h$ and $2Bh.5$, respectively (see Fig. 2.10) [1–10].

To transform a bit address represented by $2qh.r$ into a hexadecimal address xy_h , we must operate in an opposite way as previously performed, as follows [1–10]:

- First, we must identify the q and r values by using Eq. (2.1). Afterwards, q must be transformed into decimal, multiplied by 8, and added to r , which represents the value of the address in decimal. Finally, we must transform this value into hexadecimal.

To illustrate, consider the bit address $2Ah3$. Based on Eq. (2.1), q and r are equal to A_h and 3_h . Transforming Ah into decimal, we have q equal to 10_{10} . Multiplying 10_{10} by 8 and adding 3 to it, we have 83_{10} . Finally, transforming this value into hexadecimal, the bit address is equal to 53_h . Therefore, the two representations of the bit address are $2Ah3$ and $53h$, respectively (see Fig. 2.10) [1–10].

The main use of this range of data memory is to exclusively store digital control variables of a computer system, such as machine turn-on/turn-off, activation/deactivation sensor, etc. [1–10].

Furthermore, these bits can be tested by the bit-testing instructions, such as *JB bit, jump address* (jumps to *jump address* if the bit is equal to 1) and *JNB bit, jump address* (jumps to *jump address* if the bit is equal to 0) [1–10].

II.C: Range of the Data Memory of the General Purpose The range of the data memory from 30_h to $7F_h$ (8X51) and from 30_h to FF_h , for members of the Intel MCS-51 family of microcontrollers that have 256 bytes of internal memory (8X52/53), can be only accessed with 8 bits (byte). This data memory regions are usually utilized for 8-bit variables, such as counters, temperature values obtained from a sensor, etc. [1–10].

II.D: Special Function Registers (SFRs) The addressing range from 80_h to FF_h of the data memory also includes special function registers. These registers can only be accessed by direct addressing. Some of these registers are also addressable per bit. Table 2.3 shows the special function registers (SFRs) with their respective addresses [1–10].

As some SFRs can be addressed by byte and bit, for instance, the bit 0 of port P0 (address $80h$) can be represented by $P0.0$ or $80h.0$, or simply $80h$, the second least bit significant of P0 can be represented by $P0.1$, $80h.1$ or simply $81h$, and so on. The same thinking can be used for the other SFRs that are also addressed by bit (indicated by * in Table 2.3) [1–10].

Table 2.4 shows the SFRs only available for the 8X52/53 members [1–10].

Table 2.3 The special function recorders (SFRs)

Registers	Address	Register's name
A or ACC	E0 _h	Accumulator (most of instructions are executed on it)
B*	F0 _h	Register B (must be used for multiply and division operations)
DPL	82 _h	Data pointer: least significant byte of the address of the external data memory
DPH	83 _h	Data pointer: most significant byte of the address of the external data memory
IE*	A8 _h	Interrupt enable: enable/disable the 8051 microcontroller's interrupt sources (externals, Timers/Counters, serial communication interface)
IP	B8 _h	Interrupt priority: sets the priorities of the interruption sources
SCON*	98 _h	Serial control: program/configure the operation mode of the serial communication interface
SBUF	99 _h	Serial buffer: there are two registers with this same name. One is responsible for storing a byte received by the serial communication interface ($SBUF_{in}$), and the function of the other is to store a byte that must be transmitted by the serial communication interface ($SBUF_{out}$)
PSW*	D0 _h	Program status word: responsible for defining the status of the result of an operation performed in the arithmetic and logic unit (ALU). Its bits are changed by executing the arithmetic and logic instructions. Besides, it presents the parity of the accumulator register (A or ACC). In this case, the microprocessor does not need to use the ALU
PCON	87 _h	Power control: it is responsible for programming/configuring the power control of the microcontroller
TCON*	88 _h	Timer control: responsible for showing the status of the Timers/Counters and external interruptions and programming/configuring the external interruptions
TMOD	89 _h	Timer mode: responsible for programming/configuring the operation modes of the Timers/Counters
TH0	8C _h	Counting register of the Timer/Counter 0: most significant byte
TL0	8A _h	Counting register of the Timer/Counter 0: least significant byte
TH1	8D _h	Counting register of the Timer/Counter 1: most significant byte
TL1	8B _h	Counting register of the Timer/Counter 1: least significant byte
P0*	80 _h	Special function register of the port 0 (P0)
P1*	90 _h	Special function register of the port 1 (P1)
P2*	A0 _h	Special function register of the port 2 (P2)
P3*	B0 _h	Special function register of the port 3 (P3)

*It means that the register can be addressed by byte and bit.

Table 2.4 Other SFRs available for the 8X52/53 members

Symbol	Address	Register's name
T2CON*	C8 _h	Timer/counter control register 2: responsible for programming/configuring and showing the status of the Timers/Counters 2
TH2	CD _h	Register counting of the Timer/Counter 2: least significant byte
TL2	CC _h	Register counting of the Timer/Counter 2: most significant byte
RCAP2H	CB _h	Register capture of the Timer/Counter 2: most significant byte
RCAP2L	CA _h	Register capture of the Timer/Counter 2: least significant byte

*It means that the register can be addressed by byte and bit

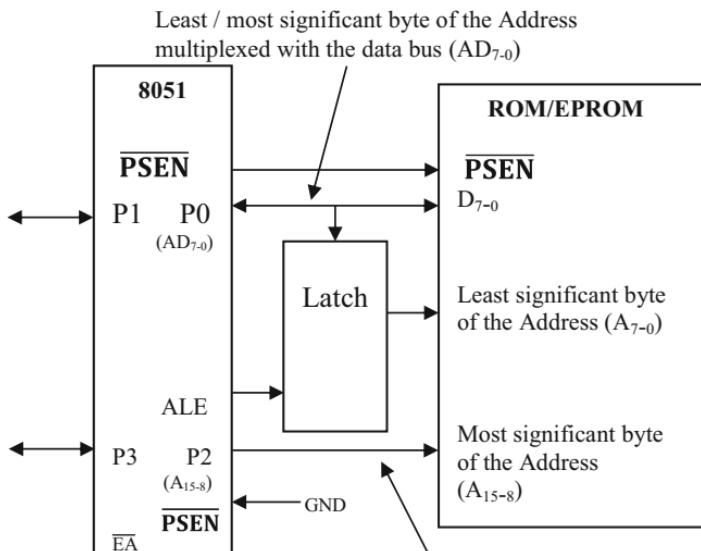


Fig. 2.11 Simplified electrical representation between the microcontroller (8051) and an external program memory

2.6 Interface with the External Memory

Figure 2.11 shows the simplified electrical representation between the microcontroller (8051) and an external program memory [1–10].

Observe that ports 0 and 2 (16 bits) are dedicated to the address with 16 bits. The external program memory and port 0 also have the function of latching the data that will be read from the external program memory. Besides, it is responsible to define the byte of the instruction to be recorded in the external program memory during its electrical programming. Port 0 (P0) presents two pieces of information along the instruction cycle (fetch and execution cycles), i.e., the least significant addresses of the external program memory to be accessed and the data bus that flows through it. This bus (P0) is represented by AD₇-AD₀ or AD₇₋₀, where A means address and D means data. Firstly, during the fetch cycle, this bus (AD₇₋₀) is set with the least

significant address of the external program memory through the least significant program counter (PCL) register. During the time that P0 defines the least significant address of the external program memory, the microprocessor sets a logic 1 in the synchronization control signal named address latch enable (ALE) to perform the demultiplexing process (capture) of this information (A_{7-0}) through an external D-type flip-flop or latch. The output of the D-type flip-flop or latch must be connected to the A_{7-0} of the external program memory. Port 2 (P2) has the function of defining the most significant address during the instruction cycle (fetch and execution cycles). This value is defined by the most significant byte of the program counter (PCH). After some time, P0, which was operating as an address bus (an output bus), becomes a data bus (an input bus), and the ALE control signal is set to logic 0. Thus, after the read control signal (\overline{PSEN}) activated in logic 0 by the microprocessor (read operation), the data (D_{7-0}) contained in the external program memory is latched by the data bus (P0). This process occurs during the execution cycle [1–10].

Figure 2.12 illustrates a simplified electrical representation between the microcontroller (8051) and an external data memory of 2 Kbytes. In this case, the CPU is running a piece of software that is stored in the internal ROM because the \overline{EA} is connected to the V_{CC} . Port 0 acts as the least significant address and data bus (AD_{7-0}). These two pieces of information are multiplexed in P0, and the three (3) bits of port 2 (P2.2-P2.0) must be defined as output bits to define the values of A_{10-8} to page the RAM in every 256 bytes. This must be done because the \overline{EA} is connected to the V_{CC} to run the software that needs to be stored/recorded in the

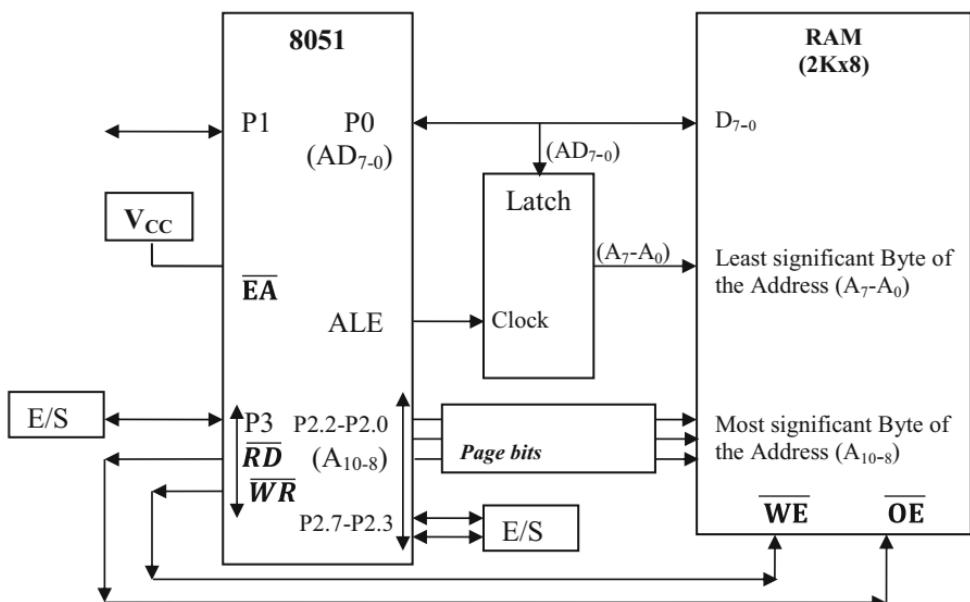


Fig. 2.12 Simplified electrical representation between the microcontroller (8051) and an external data memory

internal program memory. Therefore, P2 is not able to define the PCH to address the external data memory. The CPU generates read **RD** (P3.7) and write **WR** (P3.6) control signals to access the external RAM. The rest of the bits not used in this hardware, such as the bits of the P1, P2, and P3, can be used as input/output bits [1–10].

There can be up to 64 Kbytes of external data memory. The addressing of the external data memory can be 1 or 2 bytes. The 1-byte address is generally used by combining one or more input and output lines to page the RAM address in every 256 bytes. The 2-byte address can also be used, since the **EA** is equal to logic 0, and in this case, the eight most significant bits of the address bus are defined by port 2 [1–10].

2.7 Mapping Memory and I/O ICs Mapped as Memory

Each content of an external memory (program or data) must be only correlated with one single address. If the hardware designer does not respect this important design rule and the same address accesses two contents of the external memory during a read operation, they can define different data in the data bus (if they have different values in their contents), and consequently a short-circuit in the bits with different logic levels will occur. Therefore, this rule must be complied by the hardware designers.

Additionally, if the microcontroller is externally connected to different integrated circuits (IC), the hardware designer must always select/enable only one single IC at a time. Besides, each content of the memory or registers of these ICs must be only one single address, as previously discussed. Besides, all ICs that are connected to a microprocessor/microcontroller must present a pin named “chip enable” (**CE**) or “chip selected” (**CS**) to which they are responsible to enable (**CS=0 or CE=0**) the access (read and write operations). If the **CE pins** of these ICs are not activated (**CS=1 or CE=1**), their data buses are in three-state conditions (open circuit), and therefore they cannot be accessed through the data bus of the microprocessor/microcontroller [1–10].

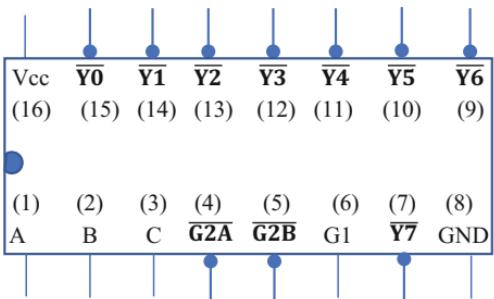
The typical technique used to generate a control signal for the chip enables or chip selects of the ICs is performed by using a decoder (combinational circuit that enables a single output, depending on the binary combination of its inputs). Thus, the inputs of the decoder must be connected to some bits of the address bus of the microprocessor/microcontroller, usually some of the most significant bits, depending on the quantity of addresses which will be necessary to address each content of the memory. Consequently, regarding a binary combination of these address bits, only one output of the decoder will be activated. The rest of the address bits not used in the inputs of the decoder, usually the least significant bits of the address bus, are used in the addressing pins of the ICs to be accessed. The 74X138 commercial decoder (X depends on the technology used, i.e., if X is equal to C, it means that the decoder was implemented with CMOS) is widely used to perform the decode of the address bus to generate the chip selects/chip enables of the ICs, according to Table 2.5 [10].

Table 2.5 Truth table of the 74X138

Inputs						Outputs							
$\overline{G2A}$	$\overline{G2B}$	G1	C	B	A	$\overline{Y0}$	$\overline{Y1}$	$\overline{Y2}$	$\overline{Y3}$	$\overline{Y4}$	$\overline{Y5}$	$\overline{Y6}$	$\overline{Y7}$
1	X	X	X	X	X	1	1	1	1	1	1	1	1
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	0	X	X	X	1	1	1	1	1	1	1	1
0	0	1	0	0	0	0	1	1	1	1	1	1	1
0	0	1	0	0	1	1	0	1	1	1	1	1	1
0	0	1	0	1	0	1	1	0	1	1	1	1	1
0	0	1	0	1	1	1	1	1	0	1	1	1	1
0	0	1	1	0	0	1	1	1	1	0	1	1	1
0	0	1	1	0	1	1	1	1	1	1	0	1	1
0	0	1	1	1	0	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	1	1	1	0

X: may be logic 0 or 1

Fig. 2.13 The 74X138 decoder

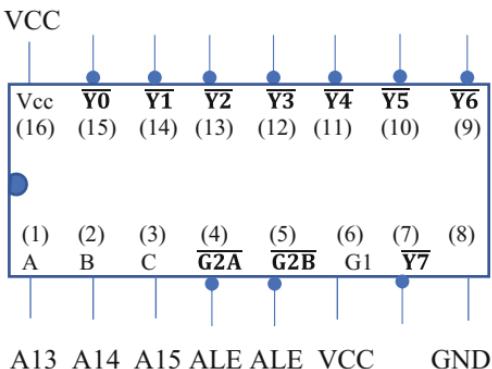


In Table 2.5, $\overline{G2A}$, $\overline{G2B}$, and G1 are the enable inputs (responsible for activating the decoder function of the IC: $\overline{G2A}=0$, $\overline{G2B}=0$, and $G1 = 1$, unless at least one of these selection inputs are not activated ($\overline{G2A}=1$ or $\overline{G2B}=1$ or $G1 = 0$), all outputs (from $\overline{Y0}$ to $\overline{Y7}$) are in logic 1, and therefore no chip enable/selected is generated), and A, B, and C are the data input of the decoder. To illustrate, if $\overline{G2A}=0$, $\overline{G2B}=0$, and $G1 = 1$ and C, B, and A are equal to 0, 0, and 0, $\overline{Y0}$ will be activated in negative logic ($=0$), and thus the IC connected to this output will be enabled to access (read and write operations) [10].

The schematic representation and the pins of a 74X138 commercial decoder are illustrated in Fig. 2.13 [10].

To illustrate, consider a 74X138 decoder used to generate different chip selects/enables to be used to address the external data memory of a 8Kx8 decoder. The most significant bits of the 8051 microcontroller's address bus (A_{15} , A_{14} , and A_{13}) are

Fig. 2.14 Example of an application of the 74X138 decoder



connected to the inputs of the 74X138 (A, B, and C). Besides, the ALE control signal of the 8051 microcontroller is used to enable the decoder to operate according to the function of the most significant bits of the decoder (A_{15} , A_{14} , and A_{13}). Besides, consider that the rest of the address bus of the microcontroller (A_{12} , A_{11} , ..., A_0) is used to address each content of the external data memory, i.e., they must be connected to the address bus of the external data memory (A_{12} , A_{11} , ..., A_0), as shown in Fig. 2.14.

Regarding the circuit schematic diagram illustrated in Fig. 2.14, the addressing map of the external data memory is shown in Table 2.6 [10].

Regarding Table 2.6, we can observe [10]:

1. The output $\overline{Y_0}$ is activated in logic 0 whenever the addresses in the range of 0000h to 1FFFh are set in the address bus of the microcontroller. The output $\overline{Y_1}$ is enabled in logic 0 when the addresses from 2000h to 3FFFh are defined in the microcontroller address bus and so on.
2. Each output of the decoder, when enabled, defines a range of 8 Kbytes of addressing of the data memory because the bits A_{15} , A_{14} , and A_{13} are connected to the C, B, and A inputs of the decoder. Therefore, each value defined by the 8051 microcontroller's address bus is capable of activating only one decoder output and also addressing only one content of the external data memory. Therefore, we can say that the 74X138 decoder configured, as illustrated in Fig. 1.14, is able to divide the 64 Kbytes addressing, which the 8051 microcontroller's address bus is capable of generating in eight parts, and each part is able to address 8 Kbytes of external data memory.
3. In order to know which is the decoder output enabled as a function of the value of the address bus defined by the 8051 microcontroller, the following procedure must be done:
 - Firstly, identify the connections between the address bus of the 8051 microcontroller and the inputs of the decoder.
 - Transform the value of the address bus into binary.

Table 2.6 Mapping of the external data memory addressing performed by using the 74x138 decoder

Decoder			Address bus of the external data memory												Addresses in hexadecimal	Decoder Outputs	Range of addresses		
C	B	A	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀				
Microcontroller Address Bus																			
A ₁₅	A ₁₄	A ₁₃	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	Initial/Final Address	CS		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000H	$\overline{Y_0}$	8 K
0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_0}$	
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1FFFH	$\overline{Y_0}$	
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2000H	$\overline{Y_1}$	8 K
0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_1}$	
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFFH	$\overline{Y_1}$	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000H	$\overline{Y_2}$	8 K
0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_2}$	
0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	5FFFH	$\overline{Y_2}$	
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6000H	$\overline{Y_3}$	8 K
0	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_3}$	
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7FFFH	$\overline{Y_3}$	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8000H	$\overline{Y_4}$	8 K
1	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_4}$	
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	9FFFH	$\overline{Y_4}$	
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A000H	$\overline{Y_5}$	8 K
1	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_5}$	
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	BFFFH	$\overline{Y_5}$	
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C000H	$\overline{Y_6}$	8 K
1	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_6}$	
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	DFFFH	$\overline{Y_6}$	
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	E000H	$\overline{Y_7}$	8 K
1	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	:	$\overline{Y_7}$	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFH	$\overline{Y_7}$	
1 st Most significant address			2 nd Most significant address			2 nd Least significant address			1 st Least significant address										

- Based on the connections performed between the address bus of the 8051 microcontroller and the inputs of the decoder, obtain the decoder output that is activated.
- With the value of the rest of the bits of the address bus, get the address of the content of the external data memory. For instance, regarding Fig. 2.14, if the address bus of the 8051 microcontroller is equal to 912Dh, the decoder output activated is $\overline{Y_4}$ because of the bits A₁₅, A₁₄, and A₁₃ being connected to the C, B, and A inputs of the decoder. Besides, the address of the content of the

external data memory that will be accessed is defined by the bits A_{12}, A_{11}, \dots , and A_0 of the 8051 microcontroller's address bus, which in this case is equal to $1000100101101_2 = 112D_h$ and corresponds to the value in decimal of $1.16^3 + 1.16^2 + 2.16^1 + 13.16^0 = 4141_{10}$. Similarly, if the address bus of the 8051 microcontroller is defined to $E999_h$, the decoder output activated is $\overline{Y7}$, and the address of the content of the external data memory that will be accessed is equal to $0100110011001_2 = 0999_h$, which corresponds to the value in decimal of $0.16^3 + 9.16^2 + 9.16^1 + 9.16^0 = 2457_{10}$, according to Table 2.7.

- By connecting, for example, the decoder output $\overline{Y0}$ to the chip select/enable of an 8Kx8 memory, its addresses range from 0000h to 1FFFh. By connecting, for instance, the decoder output $\overline{Y1}$ to the chip select/enable of another 8Kx8 memory, its addresses range from 2000h to 3FFF, and so on.
- By connecting, for example, the decoder output $\overline{Y6}$ to the chip select/enable of a 2Kx8 external data memory, which presents only 11 bits ($2K = 2048 = 2^{11} = 2^A$, therefore $A = 11$ bits, where A is the quantity of the address bits, according to Chap. 1 of this book), its addresses can be obtained as described in Table 2.8.

Observing Table 2.8, the address bits A_{12} and A_{11} of the 8051 core microcontroller's address bus are not used to address the 2Kx8 external data memory because $A_{15}-A_{13}$ are used to generate the chip select/enable through the 74x138 decoder and the $A_{10}-A_0$ are used to address each content of the external data memory. Thus, we have $2^2 = 4$ possibilities to address this memory, i.e., when they are $00_2, 01_2, 10_2$, and 11_2 , respectively. Therefore, we have four possibilities to address this 2Kx8 external data memory. These ranges of addressing are $C000_h-C7FF_h$, $C800_h-CFFF_h$, $D000_h-D7FF_h$, and $D800_h-DFFF_h$. It means that the 8 Kbytes of addressing by the $\overline{Y6}$ is divided into four parts of 2 Kbytes, and then we have four different ranges of addressing for this memory [10].

Table 2.7 Obtaining the decoder output (chip select/enable) activated and the address of the content of the external data memory that will be accessed by the microcontroller

Decoder			Address bus of the external data memory															Addresses in hexadecimal	Decoder Outputs
C	B	A	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0				
Microcontroller Address Bus																			
A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Initial/Final Address	\overline{CS}		
1	0	0	1	0	0	0	1	0	0	1	0	1	1	0	1	912D _h	$\overline{Y0}$		
1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	1	E999 _h	$\overline{Y7}$		

Table 2.8 Obtaining the addresses of the content of the external data memory that will be accessed by the microcontroller

Decoder			Address bus of the external data memory															Addresses in hexadecimal	Decoder Output	Range of address
C	B	A	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀					
Microcontroller Address Bus																				
A ₁₅	A ₁₄	A ₁₃	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	Initial / Final Address	CS			
1	1	0	X	X	0	0	0	0	0	0	0	0	0	0	0	0	?	Y ₆		
If XX=00 ₂ then the range of address is equal to:																				
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C000 _h	Y ₆	2K	
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:			
1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	C7FF _h	Y ₆		
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	C800 _h	Y ₆	2K	
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:			
1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	CFFF _h	Y ₆		
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	D000 _h	Y ₆	2K	
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:			
1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	D7FF _h	Y ₆		
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	D800 _h	Y ₆	2K	
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:			
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	DFFF _h	Y ₆		

This same approach, named input/output ICs mapped as memory, can be used to generate the chip select/enable of these devices [10].

2.8 Reset (Initialization) Signal

The control signal of the reset of the MSC-51 microcontroller family (8051 core) is activated when pin 9 remains at a high level for at least two machine cycles. The circuit for generating this electrical signal is illustrated in Fig. 1.4 of Chap. 1. The typical values for the resistor and the capacitor of this are 10 KΩ and 10 µF, respectively. When the microprocessor system is energized, the capacitor is de-energized, and therefore the voltage across the resistor is equal to the supply voltage (V_{CC}). After a time of $5.R.C = 5.10 \mu F.10K = 0.5$ s, for example, the capacitor voltage becomes approximately equal to the V_{CC} , and consequently the voltage in the resistor becomes zero [1–10].

During the presence of the reset signal, some special function registers (SRFs) are initialized in the CPU [1–10]:

- The program counter (PC) is set to 0000_h . Since the function of the special function register (SFR) named program counter (PC) is to store the address of the next instruction to be fetched and executed (instruction cycle) by the microprocessor of the microcontroller, when the reset signal occurs, the microprocessor will run the instruction located in the program memory whose address is 0000_h (ROM / EPROM). Thus, the software designer that is responsible for the firmware must store the first instruction of the firmware from the address 0000_h of the internal program memory. Besides, if there is an external program memory, the hardware designer must always map it from address 0000_h , and the software designer should always store/record the firmware from address $0000h$ in this external program memory.
- The special function register stack pointer (SP) is initialized with the value 07_h . All data stored in RAM through instructions using the stack will be stored from memory address 08_h because of the instructions that are using the SP. First, they increment one unit in the SP, and then they store a byte in the content of the address of the memory whose address is given by the content of the SP. The instructions that use the SP are PUSH, POP and LCALL, ACALL, RET, and RETI.
- The ports (P0, P1, P2, and P3) will be set to FF_h , i.e., all of them will be programmed/configured as inputs.
- The special function register of the serial communication interface (SCON) is set to 00_h .
- The five least significant bits of the SFRs interrupt enable (IE) and interrupt priority (IP) are set to 00_h .

2.9 The Clock Signal

All members of the MCS-51 microcontroller family (8051 core) have an internal oscillator that can be used as a clock signal for the microprocessor (CPU). Thus, to use it as an internal oscillator, a crystal or a ceramic resonator must be connected between the pins named XTAL1 and XTAL2 of the microcontroller, according to Fig. 1.13 of Chap. 1 [1–10].

The internal clock signal generator determines the sequence of states that defines the machine cycle of the microcontroller [1–10].

2.10 The Machine Cycle

The machine cycle of the MCS-51 from INTEL consists of six states, numbered from S1 to S6. The time of each state corresponds to two clock periods. A machine cycle has 12 clock periods. Therefore, if a crystal whose frequency (f_{clock}) is equal to 12 MHz is used, the machine cycle has a period of 12 MHz/12, i.e., equal to 1 MHz, and consequently the period of the clock ($T_{clock} = 1/f_{clock}$) is equal to 1 μs . Each

machine cycle is divided into two half-phases (phase 1 and phase 2), each one corresponding to the clock period [1–10].

If access to the external data memory occurs, two pulses will not exist because of the address and data buses that are being used for accessing the data memory through the MOVX instruction. In this way, the machine cycle regarding the data memory access is twice as large than the one observed in the program memory [1–10].

When the microprocessor (CPU) is running a program from the internal program memory, the clock is not activated, and the addressing of the external program memory is not issued. However, the ALE signal continues being activated twice per machine cycle [1–10].

Therefore, it can be used as a clock output signal to be applied in other ICs if necessary. To illustrate, regarding a crystal whose clock is equal to 12 MHz, each clock signal presents a f_{cristal} equal to $12 \text{ MHz}/12 = 1 \text{ MHz}$. Since the ALE control signal is formed by three clock signals, the ALE can be used as a clock signal of 333.33 KHz [1–10].

2.11 Execution of Instructions Step-by-Step

The MCS-51 microcontrollers' family (8051 core) from Intel has an interrupt system structure which allows it to run pieces of software step-by-step with minimum effort. This is possible because the request of an interruption is not handled by the microprocessor while an interrupt source of the same priority level is still being executed or while the return interruption (RETI) instruction has not yet been executed. Besides, whenever an interrupt source is answered, it cannot be retried until a main program instruction is executed. Therefore, one way to use this feature of running a software step-by-step is to program/configure external interrupt sources (**INT0**, bit 2 of P3 or P3.2 or **INT1**, bit 3 of P3 or P3.3) to be level-enabled. Its interrupt source attendance subroutine must be ended with the following coding if we use the **INT0** [1–10]:

```
JNB    P3.2,$    ; Wait for the bit P3.2 (INT0) to change its logic level from 0 to 1  
JB     P3.2,$    ; Wait for the bit P3.2 to change its logic level from 1 to 0  
RETI           ; Return to the main program
```

Consider an external interface with a switch, for instance, whose its output must be connected to P3.2 of the 8051 microcontroller (external interruption 0, **INT0**). Initially, the output of this interface must be at the low logic level. Thus, the **INT0** will be activated (switch deactivated) and the CPU will attend this interrupt source 0 (**INT0**). Afterwards, the CPU will run its corresponding attendance subroutine (jump to the address 0003_h; address where we must store this attendance subroutine, which is defined by the manufacturer, according to Fig. 2.4) and therefore it will execute the instruction JNB P3.2, \$ (if bit 2 of P3 is equal to 0, the CPU will jump to the address of this own instruction) as indicated above. Therefore, the

microprocessor will continue running this instruction (JNB P3.2, \$) until P3.2 changes to a high logic level through the activation of the switch of the external interface. Then, the CPU will run the instruction JB P3.2,\$, and it will wait for the logic level of P3.2 to change from 1 to 0. This occurs when the user deactivates the switch of the interface. After that, the RETI instruction will be run by the microprocessor of the 8051 microcontroller, and its CPU will return to the attendance subroutine of the INT0, and consequently it will run one instruction of the main program (software that we desire to run step-by-step). So, the CPU runs a single instruction, and as P3.2 is again at a low logic level because of the external interface, a new INT0 immediately occurs. In this way, the CPU executes the interrupt source service subroutine 0 again (INT0), and all processes previously described occur again. One instruction of the main program is executed at a time, always when P3.2 changes its logic level from 0 to 1 (activation of the switch) and from 1 to 0 (deactivation of the switch).

2.12 Low Power Operation

During the normal operation of the microcontroller, the internal RAM consumes its power from the power supply (V_{CC}). However, if the voltage in the reset/ V_{PD} pin exceeds the V_{CC} , it becomes the power source for the internal RAM. This allows a secondary power source (backup) to be used to retain the RAM data in a V_{CC} power source failure event [1–10].

In order to take advantage of this feature, the system developed by a user can perform the fault detection in the power source (V_{CC}), interrupting the CPU to transfer the relevant data from the RAM, and another secondary power supply (backup) must be activated in the RST\VPT pin. When the power supply (V_{CC}) returns to its normal condition, the V_{PD} would need to be deactivated through a reset signal [1–10].

2.13 The Power Reduction Mode

In this version, two modes of power reduction are presented: Idle and low power [1–10].

2.13.1 The Idle Mode

In the Idle mode, the oscillator works only for the external interrupt sources, Timers/Counters and serial communication interface. In this mode, the oscillator for the CPU is disabled, and there is no clock signal for it [1–10].

The CPU conditions of the special functions registers are preserved with respect to the contents of the stack pointer (SP), program counter (PC), program status word register (PSW), accumulator (A or ACC), and all other SFRs [1–10].

There are two modes to end the Idle mode [1–10]:

- When any interrupt source is enabled, since it has previously been enabled. The PCON.0 (Idle mode enabler bit of PCON) is reset by the hardware. Therefore, the interrupt will be attended, and then the CPU will run the RETI instruction, and finally the system will enter the Idle mode again.

The flags GF1 and GF0 of PCON can be used to indicate whether the sources of interruption have occurred during the normal or Idle operation modes:

- When a hardware reset signal is generated.

2.13.2 The Low Power Mode

In the low power mode, the internal oscillator is deactivated, and all functions are stopped, but only the internal RAM is energized and remains in operation. Besides, the special function registers (SFRs) are not energized. The only way to get out of the low power mode is through a hardware reset signal [1–10].

The V_{CC} can be reduced to minimize power consumption. Care should be taken to ensure that the V_{CC} is not reduced before the low power mode is activated and the V_{CC} is restored to its normal operating level before the low power mode is terminated [1–10].

The reset signal is what terminates the low power mode and releases the oscillator operation. The initialization signal must not be activated before the V_{CC} is recovered [1–10].

The Idle and low power modes are activated by the bits belonging to the special function register “power control register (PCON) (SFR address equal to 87h), which is also addressable per bit [1–10].

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
(PCON)=	SMOD	–	–	–	GF1	GF0	PD	IDL

Where:

- SMOD (serial mode) is the bit that enables the dual baud rate of the serial communication interface through the Timer/Counter 1. When it is equal to 1 logic, the baud rate is doubled, i.e., the frequency of transmitting and receiving the serial communication interface is doubled, when programming the serial communication channel is in modes 1, 2, or 3.
- GF1 is the general purpose flag 1.
- GF0 is the general purpose flag 0.

- PD (power down) means that making this bit equal to logic 1, the low power mode is activated.
- IDL (Idle mode) means that making this bit equal to logic 1, the Idle mode is activated.

2.14 Solved Exercises

1. Consider that you are a hardware designer of a computer system and you must use one microcontroller from the MCS-51 family (Intel). It is known that your prototype needs an internal program memory of 2 Kbytes in order to manage 200 control variables, 5 interrupt sources, 2 Timers/Counters, and a total of 27 actuators and sensors. What member of this microcontrollers' family should you use?

Solution According to Table 2.1, one possibility is to use the 8052AH microcontroller because it has 8 Kbytes. It will use only 2 Kbytes of the program memory and 256 volatile memory (RAM) bytes which can store the 200 control variables. Besides, it presents three Timers/Counters and only two of them will be used. It has 32 input/output ports and only 27 of these bits will be used.

8052AH	8032AH	8752BH	8K	256	4	3	X			8/6
--------	--------	--------	----	-----	---	---	---	--	--	-----

It is important to highlight that the 200 control variables are related to the internal data memory, and the 27 actuators and sensors are related to the number of input/output bits of the ports.

2. Consider the bit address $4C_h$. Where is this bit located in the data memory (memory address and bit position)? What is another way of representing this bit address?

Solution By applying the procedure described in Sect. 2.5.1, II.B:

Transform the address $4C_h$ into decimal: $4 \cdot 16^1 + 12 \cdot 16^0 = 76_{10}$.

Divide the address value into decimal (76_{10}) by 8 ($76/8$) in order to find the q and r:

$$q = 9_{10} \text{ and the rest } r = 4_{10}.$$

Transform the quotient (q) and the rest (r) into hexadecimal: $q = 9_{10} = 9_h$ and $r = 4_{10} = 4_h$.

Apply the formula: $4Ch = [20h + q_h]h.r = [20h + 9_h].h4 = 29h.4$, i.e., the address of bit $4Ch$ is equal to $29h.4$ (see Fig. 2.10).

3. Consider that the contents of the special function register (PSW) have been initialized with the value E3h. Which bank of registers was selected?

Solution

bits	7	6	5	4	3	2	1	0
(PSW) =	C	AC	F0	RS1	RS0	OV	-	P
E3 _h =	1	1	1	0	0	0	1	1

As RS1 and RS0 are equal to 00₂, the selected bank was 0, and the addresses of the registers R0, R1, . . . , and R7 are 00_h, 01_h, 02_h, . . . , and 07_h, respectively.

2.15 Proposed Exercises

- 2.15.1. What is the first member of the MCS-51 family of microcontrollers from Intel and what are its main characteristics?
- 2.15.2. Using Table 2.1, select a member of the MCS-51 family from Intel that best meets a prototype of a computer system that requires 4 Kbytes of ROM, 96 bytes of RAM, and low power consumption.
- 2.15.3. Describe the pins from 32 to 39 of the 8051 microcontroller.
- 2.15.4. Describe the function of the control signal \overline{PSEN} of microcontrollers of the MCS-51 family from Intel.
- 2.15.5. How much non-volatile and volatile memory can be addressed through an 8051 microcontroller? What is the non-volatile and volatile memory organization of the 8051 microcontroller?
- 2.15.6. What are the non-volatile memory addresses which are reserved to the reset signal and interrupt sources? Besides, describe what happens with a computer system when a reset signal occurs.
- 2.15.7. What is the space in bytes defined by the manufacturer that is reserved to store each service routine for the interruptions sources regarding the 8051 microcontroller? If a service routine of an interrupt source is larger than the available space by the manufacturer, how must you allocate it in the non-volatile memory?
- 2.15.8. What is the electrical schematic diagram of the hardware to access the external program memory? Explain the function of each control signal and components used to do this.
- 2.15.9. What is the electrical schematic diagram of the hardware to access the external data memory? Explain the function of each control signal and components used to do this.
- 2.15.10. What is internal data memory structure of the 8051 microcontroller? Explain each memory part.
- 2.15.11. Given the addresses of bits of the internal RAM memory, calculate another way of representing them:
I- 2Ch, II- 2Fh.6, III- 71h, and IV- 24.6

- 2.15.12. What are the special function registers of the 8051 and 8052 microcontrollers? Describe their functions.
- 2.15.13. What is the function of a decoder in a computer system that uses microcontrollers, external memories (non-volatile and volatile), and input/output devices?
- 2.15.14. Draw the block diagram (8051, latch, decoder, control signals, etc.) of a computer system consisting of an 8051 microcontroller, two external ROMs of 4Kx8 and one RAM of 32Kx8.
- 2.15.15. Design a memory mapping to divide the addressing space of an 8051 microcontroller into portions of 16Kx8. Draw the interconnections of the decoder with the 8051 microcontroller regarding the control signals needed to implement this.
- 2.15.16. What is the electric circuit of the reset signal to be connected to the reset pin of the 8051 microcontroller? What happens with the computer system when a reset signal is generated?

References

1. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
2. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
3. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
4. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
5. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
6. Atmel Corporation (1997) AT89 Series Hardware Description (0499B–B), Dec 1997
7. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
8. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D–MICRO), June 2008
9. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
10. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

8051 Microcontroller Instruction Set of the 8051 Core

3.1 Introduction

The instruction set of the 8051 family is directly related to a special function register called program status word (PSW). This special function register is able to provide the conditions of the processing of the last instruction performed, which is capable of changing it (only some instructions can modify it). It is used by the decision-making instructions to implement the control and management software of machines, processes, and tasks to be performed [1–10].

3.2 The Special Function Register “Program Status Word” (PSW)

Figure 3.1 illustrates the special function register “program status word” (PSW) [1–10].

It consists of four condition flags. They are responsible for reflecting the current condition of processing from the last instruction performed, which is able to change them. These bits are [1–10]:

1. *Carry-bit flag (C)*: it corresponds to the value zero or one that goes from the position of the most significant bit (bit 7) after the arithmetic instructions (addition and subtraction). If (C) becomes equal to zero (it is reset), the result of the arithmetic instruction can be represented with 8 bits, resulting in a value smaller or equal to 255_{10} . This means to say that it can be represented on an unsigned scale of values composed of 8 bits (from 0 to 255_{10}). The value of the result in decimal can be obtained by the expression: bit $7.2^7 + \text{bit } 6.2^6 + \text{bit } 5.2^5 + \text{bit } 4.2^4 + \text{bit } 3.2^3 + \text{bit } 2.2^2 + \text{bit } 1.2^1 + \text{bit } 0.2^0$. However, if (C) is equal to 1 (it is set), the result of this arithmetic instruction cannot be represented with 8 bits, but with 9 bits (a value higher than 255_{10}). In this condition, an error condition is indicated by the carry-bit, as this result cannot be represented on the unsigned scale (from

(PSW) =	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	C	AC	F0	RS1	RS0	OV	-	P

Fig. 3.1 The PSW

0 to 255_{10}). Therefore, the value of (C) must also be used to determine the value in decimal of the result with a weight of 2^8 , according to the expression: $(C).2^8 + \text{bit } 7.2^7 + \text{bit } 6.2^6 + \text{bit } 5.2^5 + \text{bit } 4.2^4 + \text{bit } 3.2^3 + \text{bit } 2.2^2 + \text{bit } 1.2^1 + \text{bit } 0.2^0$.

After logic operations, this bit is reset [$(C) = 0$] [1–10].

After a subtraction operation, if (C) is reset ($=0$), it means that the value to be subtracted (first argument of the subtraction operation) is greater or equal (\geq) to the value that we will subtract from the first value previously described (second argument of the subtraction operation). If (C) is set ($=1$), it means that the value to be subtracted (first argument of the subtraction operation) is smaller ($<$) than the value that we will subtract from the first value previously described (second argument of the subtraction operation). It is important to highlight that the subtraction operation is commonly used to perform comparisons between quantities [1–10].

Besides, it is important to say that the 8051 core does not perform a subtraction operation as humans are used to do in decimal. This operation is done by adding the contents of the accumulator [named (A) or (ACC)] to the complement of two (C_2) of the value to be subtracted. This approach (mathematics simulation of the subtraction operation) is only valid if the carry-bit flag (C) and auxiliary-carry flag (AC) are complemented (complement of one or C_1). That is, the subtraction operation can be represented according to Eq. 3.1 , where the carry-bit flag (C) is also taken into account to perform this operation. (C) must be done equal to zero if you intend to subtract only two quantities:

$$(A) \leftarrow (A) - (C) - (< \text{src} - \text{byte} >) = (A) + [(C) + (< \text{src} - \text{byte} >)]_{C_2}$$

$$\Leftrightarrow (C) \leftarrow (\bar{C}); (AC) \leftarrow (\bar{AC})$$
(3.1)

where ($<\text{src-byte}>$) may be a content of a register or memory location [1–10].

To illustrate, consider the contents of the accumulator (A) equal to 20_h , the contents of the register (R0) equal to 08_h , and the contents of the carry-bit flag equal to zero [$(C) = 0_2$]. Performing the subtraction operation SUBB A, R0, we must follow the following steps:

I. The symbolic representation of the instruction SUBB A, R0 is:

$$(A) \leftarrow (A) - (C) - (R0) = (A) + [(C) + (R0)]_{C_2} \Leftrightarrow (C) \leftarrow (\bar{C}); (AC) \leftarrow (\bar{AC})$$

II. First, you must add ($R0$) to (C); then, you must calculate the complement of two (complement of one incremented of 1) of this last result and finally add it to (A):

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	+
$(R0)=08_h=8_{10}$	0	0	0	0	1	0	0	0	
$(C)=0_b$								0	
$(R0)+(C)=$	0	0	0	0	1	0	0	0	$=08_h=8_{10}$

$[(R0) + (C)]_{C1}$	1	1	1	1	0	1	1	1	$=F7_h$
								1	+
$[(R0) + (C)]_{C2}$	¹ 1	¹ 1	⁰ 1	⁰ 1	⁰ 1	⁰ 0	⁰ 0	0	$=F8_h=-8_{10}$
$(A)=20_h=32_{10}$	0	0	1	0	0	0	0	0	+
$(A)+[(R0) + (C)]_{C2}$	¹ 0	0	0	1	1	0	0	0	$=18_h=24_{10}$

(C)

(AC)

go 1 of the position of the bit 6 [used to calculate the (OV)]

(C) and (AC) must be complemented (complement of 1) so that Eq. (3.1) is able to mathematically simulate the subtraction operation. Therefore, (C) becomes equal to 0, and (AC) becomes equal to 1. Based on this result, regarding the result of (C), it means that the first argument of the subtraction operation (A), which is equal to 20_h , is bigger or equal to the second argument of the subtraction operation (R0), which is equal to 08_h . Besides, this result ($18_h = 24_{10}$) can be represented in the unsigned scale (from 0_{10} to 255_{10}), which is considered by the 8051 core.

Furthermore, this flag is also used to convert the superior nibble of a binary number composed of 8 bits into another in the binary coded decimal (BCD) code, when the DA A instruction [decimal adjusted, where the accumulator (A) is the argument of this instruction] is executed [1–10].

The carry-bit flag (C) can be tested by the ‘jump not carry’ instructions (JNC jump address), ‘jump carry’ (JC jump address), ‘jump not bit’ (JNB PSW.7, jump address), or ‘jump bit’ (JB PSW.7, jump address). These instructions will be described in detail in this Chapter [1–10].

2. Auxiliary carry-bit flag (AC): it corresponds to the value zero or one that goes from the position indicated by bit 3 to the position of bit 4 (goes 0 or 1 from inferior nibble to the superior nibble of the byte) after the arithmetic instructions of addition or subtraction. This flag is used to convert the inferior nibble of the binary number composed of 8 bits into an inferior nibble represented in the binary coded decimal (BCD) code, when the DA A instruction (decimal adjusted) is executed [1–10].

The auxiliary carry-bit flag (C) can be tested through the instructions “jump not bit” (JNB PSW.6, *jump address*) or “jump bit” (JB PSW.6, *jump address*). These instructions will be described in detail in this chapter [1–10].

It is important to highlight that when we perform an arithmetic instruction with bytes represented in BCD codes, the result of this operation generates a byte in binary (it is not a BCD code), and thus it is necessary to convert it into a BCD code by using the DA A instruction. The procedure that the 8051 core performs to convert binary data into BCD code is [1–10]:

- I. If the inferior nibble is not a decimal value (0–9), i.e., (A–F), you must add 6 (the number 6 means adding the quantity of hexadecimal digits that are not to the decimal base), and the (AC) will always be equal to 0. Besides, you must do another analysis: if the inferior nibble is a decimal value (0–9), you must verify the value of the (AC). If the (AC) is equal to 1, you must add 6; otherwise you must not do this.
- II. Now you must analyze the other nibble (superior nibble) to complete the conversion from binary to BCD code. So, if the superior nibble is not a decimal value (0–9), i.e., (A–F), you must add 6, and (C) will always be equal to 0. Besides, you must do another analysis: if the superior nibble is a decimal value (0–9), you must verify the value of (C). If (C) is equal to 1, you must add 6; otherwise, you must not do this.
3. *Overflow-bit flag (OV)*: this flag is used to indicate if the result of an arithmetic operation (addition and subtraction) can be represented in the signed scale of 8 bits (from -128_{10} to $+127_{10}$), where the most significant bit of a byte defines its sign (0, positive and 1, negative), and the rest of the bits define the bit value ($6 \cdot 2^6 + \text{bit } 5 \cdot 2^5 + \text{bit } 4 \cdot 2^4 + \text{bit } 3 \cdot 2^3 + \text{bit } 2 \cdot 2^2 + \text{bit } 1 \cdot 2^1 + \text{bit } 0 \cdot 2^0$). If the (OV) is reset, it means that the result can be represented by this sign scale; however, if the (OV) is set, it means that the result cannot be represented by this sign scale (it is an error condition). This flag presents a function which is analogous to the carry-bit flag (C). The (OV) indicates an error condition regarding the signed scale, and (C) indicates an error condition regarding the unsigned scale, as indicated in Fig. 3.2 [1–10].

<i>Unsigned scale</i>											
0_{10}	1_{10}	...	126_{10}	127_{10}	128_{10}	129_{10}	130_{10}	...	254_{10}	255_{10}	
00_h	01_h	...	$7E_h$	$7F_h$	80_h	81_h	82_h	...	FE_h	FF_h	
<i>Signed scale</i>											
-128_{10}	-127_{10}	...	-2_{10}	-1_{10}	0_{10}	$+1_{10}$	$+2_{10}$...	$+126_{10}$	$+127_{10}$	
80_h	81_h	...	FE_h	FF_h	00_h	01_h	02_h	...	$7E_h$	$7F_h$	

Fig. 3.2 Illustrations of the two scales (unsigned, from 0 to 255_{10} and signed, from -128_{10} to $+127_{10}$) used by the 8051 microcontrollers family

The unsigned scale is usually used to define the variables of a computer system that only present values higher and equal to zero (positive values), such as age, weight, dimensions, etc. Additionally, the signed scale is typically utilized to specify the variables of a computer system that present negative (from 80_h to FF_h , as its most significant bit is equal to 1) and positive values (from 00_h to $7F_h$, as its most significant bit is equal to 0), such as temperature, pressure, etc. Regarding the C language, the variables are defined by the attribute “signed” and “unsigned” [1–10].

It is important to highlight that all negative numbers represented in the signed scale are represented in two’s complements (to find the complements of twos of a byte, we must invert each bit and then add 1). Therefore, in order to identify the value of a negative number of the signed scale, we must obtain its two’s complements. For instance, the negative number FF_h (11111111_2) corresponds to -1_{10} because its two’s complements is equal to 00000001_2 ; the negative number 81_h (10000001_2) corresponds to -127_{10} because its two’s complements of is equal to 01111111_2 ; etc. [1–10].

After the arithmetic operation, the overflow-flag (OV) is calculated by Expression (3.2) [1–10].

$$\begin{aligned} (OV) &= (\text{go } 0/1 \text{ of the position of the bit 7}) \text{ or-ex } (\text{go } 0/1 \text{ of the position of the bit 6}) = \\ &= (\text{C}) \text{ or-ex } (\text{go } 0/1 \text{ of the position of the bit 6}) \end{aligned} \quad (3.2)$$

The go 0/1 of the position of bit 6 is indicated in Section 3.2, 1, II (result of the subtraction operation) [1–10].

The overflow-bit flag (OV) can be tested by the instructions “jump not bit” (JNB PSW.2, jump address) or “jump bit” (JB PSW.2, jump address). These instructions will be described in detail in this chapter [1–10].

4. *Parity-bit flag (P)*: it indicates if the byte contained in the accumulator register [represented by (A) or (ACC)] presents an even parity (equal to 0 if the quantity of numbers of “1 s” of the 8 bits of the accumulator is equal to the even number, i.e., 0, 2, 4, 6, or 8) or odd parity (equal to 1 if the quantity of numbers of “1 s” of the 8 bits of the accumulator is equal to the odd number, i.e., 1, 3, 5, or 7). For instance, if (A) is equal to FF_h , its parity is equal to 0 ($P = 0$) because the quantity of “1 s” contained in this byte (FF_h) is equal to 8, which is an even number. The number 01_h presents an odd parity ($P = 1$) because the quantity of “1 s” contained in this byte is equal to 1, which is an odd number [1–10].

The parity-bit flag (P) is usually used in the serial reception/transmission of data in order to improve the consistency of the serial communication among different computer systems. It is usually used as a check bit in the serial communication [1–10].

Table 3.1 Instructions that affect the (C), (OV), and (AC) of the special function register (PSW)

Instruction	(C)	(OV)	(AC)
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	
DIV	0	X	
DA	X		
RRC	X		
RLC	X		
SETB C	1		
CLR C	0		
CPL C	X		
ANL C, <i>bit</i>	X		
ANL C, <i>/bit</i>	X		
ORL C, <i>bit</i>	X		
ORL C, <i>/bit</i>	X		
MOV C, <i>bit</i>	X		
MOV C, <i>/bit</i>	X		
CJNE	X		

Note: X can be 0 or 1

The parity-bit flag (P) can be tested through the instructions “jump not bit” (JNB PSW.0, *jump address*) or “jump bit” (JB PSW.0, *jump address*). These instructions will be described in detail in this chapter [1–10].

Besides these four bits previously described above, the (PSW) is also formed by a bit of general use (F0), which can be reset (= 0) or set (= 1) by certain instructions (SETB bit, CLR bit) and by two other bits defined by RS0 and RS1, which are responsible for selecting one of the four register banks (as described in Chap. 2, Section 2.5.2, II.A) [1–10].

It is also important to highlight that the flags carry-bit (C), auxiliary carry-bit (AC), and overflow (OV) of the (PSW) are usually changed whenever the logical and arithmetic unit (ULA) is utilized or by the instructions given in Table 3.1 [1–10].

It is important to emphasize that the general propose flag (F0) is not affected by the arithmetic and logic instructions and those indicated in Table 3.1. The parity-bit flag (P) is only affected when the content of the accumulator register [represented by (A) or (ACC)] is changed [1–10].

Therefore, whenever an arithmetic, logic, or rotation instruction is executed, according to Table 3.1, in addition to the result of the operation being stored in a certain register [usually in the accumulator, (A) or (ACC), (R0)-(R7)] or memory location, the flags (C), (OV), and (AC) of the (PSW) will simultaneously be redefined, reflecting the status/condition of the result of operation performed so that they can be used by another instruction which tests their conditions. If the special function register accumulator is the destination of the result, the parity-bit flag (P) is also defined [1–10].

5. *Zero-bit flag (Z)*: it indicates that the result of an arithmetic or logic operation is equal to zero. It is not allocated in the (PSW), but it can be tested through the instructions “jump not zero” (JNZ *jump address*) and “jump zero” (JZ *jump address*). These instructions will be described in detail in this chapter [1–10].

To illustrate the meaning of each flag of the (PSW), consider the execution of the instruction ADD A, 30h, considering the following initial conditions of the registers (A), (PSW), and memory location (30h): (A) = 3D_h, (PSW) = 00_h = 00000000₂, and (30h) = 1F_h. First, from the instruction table of the microcontroller with the 8051 core, we get the symbolic representation of this instruction: (A) \leftarrow (A) + (30h). So:

Therefore, the content of the accumulator changes its value from 3Dh to 5Ch and simultaneously the contents of the program status word register (PSW) also changes to:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
(PSW)=	C	AC	F0	RS1	RS0	OV	-	P	
	0	1	0	0	0	0	0	0	= 40 _h

Therefore, analyzing the (PSW):

(C) = 0: goes 0 of the position of bit 7 after the addition operation. It means that the result can be represented in 8 bits (from 0_{10} to 255_{10} because the result is equal to $5Ch = 92_{10} < 255_{10}$). Besides, if a decimal adjustment operation (DA A) is performed, i.e., if you convert a binary/hexadecimal number to a BCD code, the value 6 will not be added to the superior nibble.

(AC) = 1: go 1 of the position of bit 3 after the addition operation. If a decimal adjustment operation (DA A) is performed, i.e., if you convert a binary/hexadecimal number into a BCD code, the value 6 will be added to the inferior nibble.

(F0): it continues with the previous value, which is equal to 0, once this bit is not influenced by this instruction.

(RS1) and (RS0): they continue with the previous values, which are equal to 0, as they are not affected by this instruction, i.e., they continue to select the first register bank of the internal RAM.

(OV) = 0: it is calculated by go 0 of bit 7 or-ex go 0 of bit 6 = 0 or-ex 0 = 0; it means that there is not an error condition in the numerical representation of the result within the signed scale (from -128 to +127)

(P) = 0: there are four numbers of “1 s” inside the accumulator (A)=5Ch=01011100₂, and therefore its parity is even.

3.3 Addressing Modes of the 8051 Core Microcontroller Family

There are several ways to access or address the registers and memory locations of a microcontroller. The 8051 core microcontroller family has five different addressing modes. They are *by register, direct, indirect or indexed by register, immediate, and by register more indirect or indexed by register*. Table 3.2 presents the five addressing modes of the 8051 core microcontroller family with the register and memory locations [1–10].

The addressing modes indicated in Table 3.2 are classified with these designations when the accumulator register [named (A) or (ACC)] is one of the arguments of the instructions; otherwise, the instruction presents two different addressing modes, and it is defined to present a combined or mixed addressing mode. Consequently, its addressing mode is determined by the two addressing modes. For instance, consider the MOV @R0,30h instruction – since the accumulator register is not used in this instruction, it has mixed or mixed addressing mode. Therefore, its addressing mode is defined by two addressing modes: indirect or indexed by register (@R0) and direct (30h) [1–10].

3.3.1 Addressing by Register

This addressing mode works with the existing registers in the selected register bank. These instructions contain, through their machine code or object code, at least 3 bits corresponding to one of the selected registers (R0 to R7). The B, DPTR (data pointer), C (carry-bit), and A or ACC (accumulator) registers of the Boolean processor can also be addressed as registers. Some examples of this type of addressing are [1–10]:

Table 3.2 Addressing modes of the registers and related memory locations

Addressing modes	Argument (registers and memory locations)
Immediate	Program memory
By register	R0-R7 e A (ACC), B, C (<i>carry-bit</i>) e DPTR
Direct	The 128 bytes least significant of the internal RAM Internal and special function registers
Indirect or indexed by register	Internal RAM (@R0, @R1 e SP) and external data memory (@R0, @R1 e @DPTR)
Indirect by base registers or indexed by base registers	Program memory (@DPTR+A e @PC + A)

Example 1: MOV A, R1

Symbolic representation of the instruction: (A)←(R1)

Description of the instruction: the content of the register R1 will be copied to the contents of the accumulator (A) register.

Example 2: MOV R2, A

Symbolic representation of the instruction: (R2)←(A)

Description of the instruction: the content of the accumulator (A) register will be copied to the content of the R2 register.

3.3.2 Direct Addressing

This access mode specifies directly in the instruction a memory address of the internal RAM of the 8051 core microcontroller. Only the special function registers (SFRs) and the inferior 128 bytes of the internal RAM can be directly accessed by this type of addressing [1–10]. Here are some examples:

Example 1: MOV A, 30h

Symbolic representation of the instruction: (A)←(30h)

Description of the instruction: the content of the memory location whose address is 30h is copied to the contents of the accumulator (A) register.

Example 2: MOV 7Ah, A

Symbolic representation of instruction: (7Ah)←(A)

Description of the instruction: the content of the accumulator (A) register will be copied to the content of the memory location whose address is 7Ah.

3.3.3 Indirect or Indexed by Register

This addressing mode uses only the contents of the R0 and R1 registers of the selected register bank as an address (pointer) of a memory location within a block of 256 bytes. These registers are used as addresses of the inferior 128 bytes of the internal RAM and superior 128 bytes of internal RAM for those members that present 256 bytes of internal RAM (8032/8052) or as the address of the inferior 256 bytes of external RAM. The access to the 64 Kbytes of the external RAM addressing is done through a 16-bit register, i.e., with the help of the data pointer (DPTR) register [1–10].

The instructions that store data on the stack, such as PUSH <direct> and POP <direct>, also use this type of addressing [1–10].

The stack pointer (SP) register can point to anywhere in the internal RAM [1–10].

To illustrate, some examples are made with this type of addressing:

Example 1: MOV A, @R0

Symbolic representation: $(A) \leftarrow ((R0))$. Note that in this instruction, there is an @ symbol, and there are also two parentheses in the second operand.

Description of the instruction: the content of the memory location whose address is defined by the content of the R0 register is copied to the content of the accumulator (A) register.

Example 2: MOV @R1, A

Symbolic representation: $((R1)) \leftarrow (A)$

Description of the instruction: the content of the accumulator (A) register is copied to the content of the memory location of the internal RAM whose address is provided by the content of the R1 register.

3.3.4 Immediate Addressing

This type of addressing allows the use of constant values in the instructions. It is usually used to initialize a register or memory position of the internal RAM [1–10]. To illustrate, consider the examples given below:

Example 1: MOV A, #3Ah

Symbolic representation: $(A) \leftarrow \#3Ah$ (note that there is the “#” sharp sign in this statement).

Description of the instruction: the content of the accumulator (A) register will be initialized (initialization operation of the accumulator) with the constant value, which is equal to 3Ah.

3.3.5 Indirect by Base Registers or Indexed by Base Registers

This addressing mode allows 1 byte to be accessed from the program memory, which is addressed through the addition of a base register (DPTR, data pointer or PC, program counter) and the indexed accumulator (A) register. This mode especially facilitates the access to tables [1–10]. Below are some examples of this type of instructions:

Example 1: MOVC A, @A + DPTR

Symbolic representation: $(A) \leftarrow ((A) + (DPTR))$

Description of the instruction: the content of the program memory position (ROM/EPROM/EEPROM/Flash) whose address is given by the sum of the contents of the accumulator (A) register and data pointer register (DPTR) is copied to the content of the accumulator (A) register.

3.3.6 Combined or Mixed Addressing

Some examples of the type addressing mode are given below [1–10]:

Example 1: MOV R0,20h

Symbolic representation: $(R0) \leftarrow (20h)$

Description of the instruction: the content of the memory position whose address is equal to 20h is copied to the content of the R0 register. This is a combined addressing mode (mixed), by register and direct.

Example 2: MOV 30h, R4

Symbolic representation: $(30h) \leftarrow (R4)$

Description of the instruction: the content of the R4 register is copied to the memory position whose address is equal to 30h. This is a combined (mixed) addressing mode: direct and by register.

Example 3: MOV R1, # 40h

Symbolic representation: $(R1) \leftarrow \#40h$

Description of the instruction: the content of the R1 register is initialized with the constant value of 40h. This is a combined (mixed) addressing mode, by register and immediate.

Example 4: MOV 60h, #55h

Symbolic representation: $(60h) \leftarrow \#55h$

Description of the instruction: the content of the memory position whose address is equal to 60h is initialized with the constant value equal to 40h. Its address mode is combined (mixed), direct and immediate.

Example 5: MOV @R1, #33h

Symbolic representation: $((R1)) \leftarrow \#33h$

Description of the instruction: the content of the memory position whose address is given by the content of the R1 register is set with the constant value equal to 33h. Its address mode is combined (mixed), indirect or indexed by register and immediate.

Example 6: MOV 50h, @R1

Symbolic representation: $(50h) \leftarrow ((R1))$

Description of the instruction: in the content of the memory position whose address is 50h is copied with the content of the memory position whose address is given by the content of the R1 register. Its address mode is combined (mixed), direct and indirect or indexed by register.

Example 7: MOV @R0, 50h

Symbolic representation: $((R0)) \leftarrow (50h)$

Description of the instruction: in the content of the memory position whose address is given by the content of the R0 register is copied with the content of the memory position whose address is 50h. Its address mode is combined (mixed), indirect or indexed by register and direct.

Example 8: MOV 40h, 50h

Symbolic representation: (40h)←(50h)

Description of the instruction: in the content of the memory position whose address is 40h is copied with the content of the memory position whose address is 50h.

It is a combined addressing mode: direct and direct.

3.4 Instructions of the 8051 Core Microcontrollers Family

All members of the 8051 microcontroller family have the same set of instructions. These instructions can be defined by 1, 2, or 3 bytes. The instructions that this microcontroller can perform through its microprocessor are data transfer operations, arithmetic and logic operations, rotation operations, unconditional and conditional jump, call to subroutines, return to subroutines, etc. [1–10].

3.4.1 Instructions Related to the Internal RAM

Table 3.3 shows the set of instructions for accessing the internal RAM [1–10].

In Table 3.3 below, <src> is the source of the information, <dest> is the destination of the information, “date” is a constant value, <byte> can be the content of a register or the content of a memory location, and R_i is the content of the register used in the indirect/indexed addressing mode, i.e., only (R0) or (R1) [1–10].

3.4.2 Instructions Related to the External RAM

Table 3.4 shows the set of instructions related to the external RAM. Only indirect or indexed addressing can be used. You can use an addressing of 1 byte through @R_i (R0 or R1 of the selected bank) or use an addressing with 16 bits (2 bytes) through the DPTR register. The disadvantage of using a 2-byte addressing is due to the necessity to use all bits of Port 2 as an address bus. By using a 1-byte addressing, there is no the need to use all bits of Port 2 [1–10].

The read and write control signals from external RAM are only enabled during the execution of the MOVX instruction. These signals are usually inactive, and these pins can be used as extra input and output (I/O) lines [1–10].

Table 3.3 Instructions related to the internal RAM access (crystal frequency: 12 Mhz)

Mnemonic	Instruction	Address modes					Execution time (μs)		
		Direct	Indirect or indexed	By register	Immediate				
Move									
The content of the register/memory location <src> is copied to the content of the accumulator (A)									
MOV A,<src>	(A) ← <src>	X	X	X	X	1			
The content of the accumulator (A) is copied to the content of the register/memory location <src>									
MOV < dest>, A	<dest> ← (A)	X	X	X		1			
The content of the register <src> is copied to the content of the register/memory location <dest>									
MOV < dest>, <src>	<dest> ← <src>	X	X	X	X	2			
The content of the data pointer register (DPTR which presents sixteen bits) is initialized with the constant value equal to data ₁₆									
MOV DPTR, #data ₁₆	(DPTR) ← #data ₁₆				X	2			
Stores in the stack (internal RAM): the content of the stack pointer (SP) register is incremented with one unit (points to the next internal RAM location), and the content of the register / memory location <src> is copied to the content of the memory location whose address is given by the content of the (SP) register									
PUSH <src>	(SP) ← (SP) + 1; ((SP)) ← <src>	X					2		
Reads data from the stack (internal RAM): the content of the memory location whose address is given by the content of the (SP) register is copied to the content of the register / memory location <dest>, and the content of the stack pointer register (SP) is decremented by one unit (points to the memory location of the data previously stored in the stack)>									
POP <dest>	<dest> ← ((SP)), (SP) ← (SP)-1	X					2		
Exchange the content of the accumulator (A) with the content of the register / memory location <byte>									
XCH A,<byte>	(A) ↔ <byte>	X	X	X		1			
Only change the least significant 4 bits (inferior nibble) of the content of the accumulator (A) register with the least significant 4 bits of the content of the memory location addressed by the content of the register Ri (only R0 or R1 of the selected bank)									
XCHD A, @Ri	(A) _{3,0} ↔ ((R _i) _{3,0})		X				1		

3.4.3 Instructions of the Tables Manipulation

Table 3.5 shows two instructions for manipulating tables located in the program memory. If the table access is through the external program memory, then the read control signal is the **PSEN** [1–10].

The first statement of Table 3.5 can accommodate up to 256 bytes, numbered from 0 to FFh. The desired table value to be obtained must be loaded into the content

Table 3.4 Instructions related to the external RAM access (crystal frequency: 12 Mhz)

Mnemonic	Move	Address width	Execution time (μs)
The content of the memory location of external memory is copied to the content of the accumulator (A) register, which is addressed by the content of the register Ri and vice versa			
MOVX A, @Ri	(A) \leftarrow ((Ri))	8 bits	2
MOVX @Ri, A	((Ri)) \leftarrow (A)	8 bits	2
The content of the external memory location is copied to the content of the accumulator (A) register, which is addressed by the content of the data pointer (DPTR) register and vice versa			
MOVX A, @DPTR	(A) \leftarrow ((DPTR))	16 bits	2
MOVX @DPTR, A	((DPTR)) \leftarrow (A)	16 bits	2

Table 3.5 Program memory access operations (crystal frequency: 12 Mhz)

Mnemonic	Instruction	Execution time (μs)
The content of the program memory location is copied to the content of the accumulator (A), which is addressed by the sum of the contents of the accumulator (A) and DPTR/PC registers		
MOVC A, @A + DPTR	(A) \leftarrow ((A) + (DPTR))	2
MOVC A, @A + PC	(A) \leftarrow ((A) + (PC))	2

of the accumulator (A) register, and the content of the data pointer (DPTR) must be initialized with the address at the beginning of the table. The other statement (MOVC A, @A + PC) uses the content of the program counter (PC) as the address of the base table, and the table is accessed through a subroutine. First, you must initialize the contents of the accumulator with the desired value contained in the table, and the subroutine must be called. The table should begin immediately after the RET instruction of the subroutine, as indicated below [1–10].

```

MOV A, desired number
ACALL Table
:
Table: MOVC A, @A + PC
        RET
    
```

3.4.4 Instructions for Arithmetic Operations

Table 3.6 presents the arithmetic instructions available for the Intel MCS-51 family of microcontrollers [1–10].

3.4.5 Instructions for Logical Operations

Table 3.7 lists the logic instructions available for the Intel MCS-51 family of microcontrollers [1–10].

Table 3.6 Instructions for arithmetic operations (crystal frequency: 12 MHz)

		Address modes					
Mnemonic	Instruction	Direct	Indirect or indexed	By register	Immediate	Execution time (μs)	
Addition: the result of the addition operation between the contents of the accumulator (A) register and the register/memory location <byte> is stored in the content of the accumulator (A)							
ADD A, <byte>	(A) ← (A) + <byte>	X	X	X	X	X	1
Addition: the result of the addition operation between the contents of the accumulator (A) register and the register/memory location <byte> and the carry-bit (C) is stored in the content of the accumulator (A)							
ADDC A, <byte>	(A) ← (A) + <byte> + (C)	X	X	X	X	X	1
Subtraction: the result of the subtraction operation between the contents of the accumulator and the register/memory location <byte> and the carry-bit (C) is stored in the content of the accumulator (A)							
SUBB A, <byte>	(A) ← (A) - <byte> - (C)	X	X	X	X	X	1
Addition of a unit (increment): the content of the accumulator (A) is incremented by one unit. The result is stored in the accumulator (A) register							
INC A	(A) ← (A) + 1			Only (A)			1
Addition of a unit: the content of the register/memory location <byte> is incremented by one unit. The result is stored in the content of the register/memory location <byte>							
INC <byte>	(byte) ← <byte> + 1	X	X	X			1
Adding a unit: The content of the DPTR register is incremented by one unit. The result is stored in the content of the data pointer (DPTR) register							
INC DPTR	(DPTR) ← (DPTR) + 1			Only (DPTR)			2
Subtraction of a unit: the content of the accumulator (A) is decremented by one unit. The result is stored in the accumulator (A) register							
DEC A	(A) ← (A) - 1			Only (A)			1
Subtraction of a unit: the content of the register/memory location <byte> is decremented by one unit. The result is stored in the content of the register/memory location <byte>							
DEC <byte>	(byte) ← (byte) - 1	X	X	X			1
Multiplication: it multiplies the contents of registers A and B and produces a represented result by 16 bits. The least significant 8 bits of the result are stored in the content of the accumulator (A) register, and the most significant ones are stored in the content of register B							
MUL AB	The result of this multiplication of (A)*(B) is given by 16 bits. (B) ← Most significant byte (8 bits) of the result			Only (A) e (B)			4

(continued)

Table 3.6 (continued)

		Address modes				
Mnemonic	Instruction	Direct	Indirect or indexed	By register	Immediate	Execution time (μs)
	(A*B) (A) ← Least significant byte (8 bits) of the result (A*B)					
Division: it divides the content of the register accumulator (A) by the content of register B. The division quotient is stored in the content of the accumulator (A), and the rest is stored in the content of register B						
DIV AB	(A) ← Quotient of the result of (A)/(B) (B) ← Rest of the result of (A/B)			Only (A) e (B)		4
Decimal adjustment: it converts binary code into BCD (decimal encoded in binary) code						
DA A	Convert binary code in BCD code: Sums the value 6 in the four least significant bits if: (AC) = 1 or they do not belong to the decimal digits (A-F) Adds the value 6 to the most significant 4 bits if: (C) = 1 or they do not belong to the decimal digits (A-F)			Only (A)		1

3.4.6 Boolean Instructions

Table 3.8 illustrates the Boolean instructions (1-bit operations) available for the 8051 core-based microcontroller family [1–10].

3.4.7 Operations of the Unconditional Jump

The destination address of these statements is specified in the Assembly programming language by a label or by a program memory address. However, the target address works as a relative value. It is a signed byte (represented in two's complements) which is added to the content of the program counter register (PC). The scale of the jump varies depending on the instruction type (SJMP, AJMP, and LJMP), relative to the first byte following the instruction [1–10] (Table 3.9).

Table 3.7 Instructions for logical operations (crystal frequency: 12 MHz)

		Address modes					
Mnemonic	Instruction	Direct	Indirect or indexed	By register	Immediate	Execution time (μs)	
	AND: the result of the AND operation between the contents of the accumulator and the register/memory location <byte> is stored in the content of the accumulator (A)						
ANL A, <byte>	(A) ← (A) and <byte>	X	X	X	X	1	
	AND: the result of the AND operation between the contents of the register/memory location <byte> and the accumulator (A) register is stored in the content of the register / memory location <byte>						
ANL < byte>, A	<byte> ← <byte> and (A)	X				1	
	AND: the result of the AND operation between the contents of the register/memory location <byte> and the constant value (data) is stored in the content of the register / memory location <byte>						
ANL < byte>, #data	<byte> ← <byte> and #data	X				2	
	OR: the result of the OR operation between the contents of the accumulator and the register/memory location <byte> is stored in the content of the accumulator (A)						
ORL A, <byte>	(A) ← (A) or <byte>	X	X	X	X	1	
	OR: the result of the OR operation between the contents of the register/memory location <byte> and the accumulator (A) register is stored in the content of the register / memory location <byte>						
ORL < byte>, A	<byte> ← <byte> or (A)	X				1	
	OR: the result of the OR operation between the contents of the register/memory location <byte> and the constant value (data) is stored in the content of the register/memory location <byte>						
ORL < byte>, #data	<byte> ← <byte> or #data	X				2	
	Exclusive-OR: the result of the exclusive-OR operation between the contents of the accumulator and the register/memory location <byte> is stored in the content of the accumulator (A)						
XRL A, <byte>	(A) ← (A) or-ex <byte>	X	X	X	X	1	
	Exclusive-OR: the result of the exclusive-OR operation between the contents of the register/memory location <byte> and the accumulator (A) register is stored in the content of the register/memory location <byte>						
XRL < byte>, A	<byte> ← <byte> or-ex (A)	X				1	
	Exclusive-OR: the result of the exclusive-OR operation between the contents of the register/memory location <byte> and the constant value (data) is stored in the content of the register / memory location <byte>						
XRL < byte>, #data	<byte> ← <byte> or-ex #data	X				2	
	Clear/reset: the content of the accumulator (A) register is reset (clear)						
CLR A	(A) ← #00h			Only (A)		1	

(continued)

Table 3.7 (continued)

		Address modes				
Mnemonic	Instruction	Direct	Indirect or indexed	By register	Immediate	Execution time (μs)
Complement of one: the content of the accumulator is stored with the complement of one of its own content						
CPL A	(A) ← not (A)			Only (A)		1
Rotate: the content of the accumulator is rotated one bit to the left ($A_{n+1} \leftarrow (A_n)$ for $n = 0$ a 6 e $(A_0) = (C) \leftarrow (A_7)$, where n is an bit index. Ex: A_n for $n = 0$ corresponds to bit 0 of the accumulator (A_0) , A_n for $n = 1$ corresponds to the bit one of the accumulator (A_1) and so on						
RL A	$(A_{n+1}) \leftarrow (A_n)$ for $n = 0$ a 6 and $(A_0) = (C) \leftarrow (A_7)$			Only (A)		1
Rotate: the content of the accumulator is rotated one bit to the left considering the carry-bit flag (C) ($A_{n+1} \leftarrow (A_n)$ for $n = 0$ a 6 e $(A_0) \leftarrow (C)$ e $(C) \leftarrow (A_7)$, where n is a bit index						
RLC A	$(A_{n+1}) \leftarrow (A_n)$ for $n = 0$ a 6 e $(A_0) \leftarrow (C)$ and $(C) \leftarrow (A_7)$			Only (A)		1
Rotate: the content of the accumulator is rotated one bit to the right ($A_n \leftarrow (A_{n+1})$ for $n = 0$ to 6 ($A_7) = (C) \leftarrow (A_0)$, where n is the bit index						
RR A	$(A_n) \leftarrow (A_{n+1})$ for $n = 0$ to 6 $(A_7) = (C) \leftarrow (A_0)$			Only (A)		1
Rotate: the content of the accumulator is rotated one bit to the right considering the carry-bit flag (C) ($A_n \leftarrow (A_{n+1})$ for $n = 0$ to 6 and $(A_7) \leftarrow (C)$ and $(C) \leftarrow (A_0)$, where n is the bit index						
RRC A	$(A_n) \leftarrow (A_{n+1})$ for $n = 0$ to 6 and $(A_7) \leftarrow (C)$ and $(C) \leftarrow (A_0)$			Only (A)		1
Swap: The content of the least significant bits ($A_{3,0}$) is exchanged with the content of the most significant bits ($A_{7,4}$) of the contents of the accumulator (A) register						
SWAP A	$(A_{3,0}) \leftrightarrow (A_{7,4})$			S6 (A)		1

3.4.8 Jumper and Call-to-Condition Operations

Table 3.10 shows the list of jump instructions and calls the conditional subroutine [1–10].

Appendix A details all instructions of the microcontroller family with the 8051 core.

Table 3.8 Boolean operations with bits (crystal frequency: 12 MHz)

Mnemonic	Instruction	Execution time (μs)
AND: the result of the AND operation between the contents of the carry-bit (C) and bit (bit) is stored in the content of the carry-bit flag (C)		
ANL C,bit	(C) \leftarrow (C) and (bit)	2
AND: the result of the AND operation between the contents of the carry-bit (C) and complement of one of the bit (not (bit)) is stored in the content of the carry-bit flag (C)		
ANL C,/bit	(C) \leftarrow (C) and (not (bit))	2
OR: the result of the OR operation between the contents of the carry-bit (C) and bit (bit) is stored in the content of the carry-bit flag (C)		
ORL C,bit	(C) \leftarrow (C) or (bit)	2
OR: the result of the OR operation between the contents of the carry-bit (C) and complement of one of the bit (not (bit)) is stored in the content of the carry-bit flag (C)		
ORL C,/bit	(C) \leftarrow (C) or (not (bit))	2
Copy: the content of the bit is copied to the content of the carry-bit (C) and vice versa		
MOV C, bit	(C) \leftarrow (bit)	1
MOV bit, C	(bit) \leftarrow (C)	2
Clear/reset: the carry-bit flag (C)/bit content is reset		
CLR C	(C) \leftarrow #0	1
CLR bit	(bit) \leftarrow #0	1
Set: the content of the carry-bit/bit is set		
SETB C	(C) \leftarrow #1	1
SETB bit	(bit) \leftarrow #1	1
Complement of one: the content of the carry-bit/bit is stored with its complement of one		
CPL C	(C) \leftarrow not (C)	1
CPL bit	(bit) \leftarrow not (bit)	1
Conditional jump		
JC address	If (C) = 1 then (PC) \leftarrow address (jumps to the program memory address defined by <i>address</i>); otherwise the program executes the next instruction	2
JNC address	If (C) = 0 then (PC) \leftarrow address (jumps to the program memory address defined by <i>address</i>); otherwise the program executes the next instruction	2
JB bit, address	If (bit) = 1 then (PC) \leftarrow address (jumps to the program memory address defined by <i>address</i>); otherwise the program executes the next instruction	2
JNB bit, address	If (bit) = 0 then (PC) \leftarrow address (jumps to the program memory address defined by <i>address</i>); otherwise the program executes the next instruction	2
JBC bit, address	If (bit) = 1 then (PC) \leftarrow address (jumps to the address of program memory defined by <i>address</i>) and reset the bit [(bit) \leftarrow #0]; otherwise the program executes the next instruction	2

Table 3.9 presents the list of unconditional jump instructions (crystal frequency: 12 Mhz)

Mnemonic	Instruction	Execution time (μs)
SJMP address	Jumps to the program memory address $[(PC) \leftarrow \text{address}]$. The “address” is a range from -128 to $+127$ (short jump)	2
AJMP address	Jumps to the program memory address $[(PC) \leftarrow \text{address}]$. The “address” is a range from -1024 to $+1023$ (absolute jump)	2
LJMP address	Jumps to the program memory address $[(PC) \leftarrow \text{address}]$. The “address” can only take values from -65536 to $+65535$ (long jump)	3
JMP @A + DPTR	$(PC) \leftarrow (A) + (DPTR)$: jumps to the program memory address given by the sum of the contents of the accumulator and data pointer (DPTR)	2
ACALL address	Calls the subroutine located at the program memory address and stores the program memory address of the next instruction in the stack. The “address” can only assume values from -1024 to $+1023$ from the content of the program counter register (PC)	2
LCALL address	Calls the subroutine located at the program memory address and stores the program memory address of the next instruction in the stack. The address can only assume values from -65536 to $+65535$ from the content of the program counter register (PC)	3
RET	Returns from subroutine: reads an address from the content of the stack and stores it in the content of the PC	2
RETI	Returns from the service subroutine of an interrupt source: reads an address of the content of the stack and stores it in the contents of the PC	2
NOP	Does not perform any operation, only spends time (no operation)	1

3.5 Solved Exercises

- Regarding the following initial conditions, execute (run/process) the instructions theoretically, showing the registers and memory positions that are affected by the processing of such instructions and justify the result obtained considering the scales from 0 to 255 (unsigned scale) and from -128 to $+127$ (signed scale):
 $(A) = A6_{16}$; $(PSW) = 9A_{16}$; $(PC) = 1000_{16}$; $(20h) = 2E_{16}$; $(21h) = 7D_{16}$;
 $(22h) = 6B_{16}$; $(23h) = 5A_{16}$; $(24h) = 49_{16}$; $(25h) = 38_{16}$; $(26h) = 27_{16}$;
 $(R0) = 37_{16}$; $(R1) = 34_{16}$; $(R2) = FB_{16}$; $(R3) = 59_{16}$; $(R4) = 4E_{16}$; $(R5) = 72_{16}$;
 $(R6) = 3C_{16}$; $(R7) = 81_{16}$.

Table 3.10 Conditional jump operations (crystal frequency: 12 Mhz)

Mnemonic	Instruction	Address Mode				Execution time (μs)
Conditional jump						
JZ address	If $(A) = 0$ [$(Z) = 1$], then $(PC) \leftarrow$ address (jumps to the program memory address defined by address); otherwise, the program executes the next instruction			Só (A)		2
JNZ address	If $(A) \neq 0$ [$(Z) = 0$], then $(PC) \leftarrow$ address (jumps to the program memory address defined by address); otherwise, the program executes the next instruction			Só (A)		2
Decrement one unit of the < byte > and jump if the result is different of zero (nonzero)						
DJNZ <byte>, address	Decrement one unit of the content of the register/memory location and if the content of the register/memory location is nonzero (different of zero), jumps to the address $[(PC) \leftarrow$ address]	X		X		2
Compara os conteúdos de A e < byte > e salta se diferentes						
CJNE A, <byte>, address	If $(A) \neq <byte>$, then it calls the subroutine whose address is (address) $[(PC) \leftarrow$ address] and stores the return address of the service subroutine in the stack (program memory address of the subsequent instruction of the CJNE instruction A, <byte>, address); otherwise, the	X			X	2

(continued)

Table 3.10 (continued)

Mnemonic	Instruction	Address Mode			Immediate	Execution time (μs)
		Direct	Indirect	Indirect or indexed		
	program executes the next instruction after this instruction					
CJNE A, #data, address	If (A) is different from the date value (data), then it calls the service subroutine whose address is address [(PC) ← address] and stores the return address of the service subroutine in the stack (program memory address of the subsequent instruction of the CJNE instruction A, <byte>, address); otherwise, the program executes the next instruction subsequent to this instruction		X	X		2

I- ADDC A,21h

Answer Observing Appendix A, it turns out that this instruction has direct addressing. Therefore, the following information can be obtained:

ADDC A, direct	2	1	0011 0101 direct address	(PC) ← (PC) + 2 (A) ← (A) + (C) + (direct)
----------------	---	---	--------------------------	-----------------------------------------------

Its symbolic representation is given by:

- (PC) ← PC + 2 ; The contents of the program counter (PC) register are added
 ; by two units (points to the memory address of
 ; program of the next instruction to be executed by the CPU
 after that.)
- (A) ← (A) + (C) + (direct) ; It is stored in the content of the accumulator, the result of
 the addition

; operation between the contents of the accumulator
; (A) register,
; the carry-bit flag (C) and memory location whose address
; is 20h.

Consequently, the initial conditions of the registers are $(A) = A6_{16}$; $(C) = 1$, since the contents of the special function register (PSW) is equal to $9Ah = 10011010_2$, as follows:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	C	AC	F0	RS1	RS0	OV	-	P
$(PSW)=9A_{16} =$	1	0	0	1	1	0	1	0

And the content of the memory location whose address is equal to 21_{16} is equal to $7D_{16}$. So:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
$(A) = A6_{16} = 166_{10}$	11	10	11	10	10	11	11	0
$(21_{16}) = 7D_{16} = 125_{10}$	0	1	1	1	1	1	0	1
$(C)=1$								1
$(A) \leftarrow (A) + (C) + (21_{16}) =$	10	0	1	0	0	1	0	0

$= 24_{16} = 36_{10}$

Besides the content of the accumulator (A) register changing from $A6_{16}$ value to 24_{16} , the content of the program status word (PSW) register also changes, and the new values of the (C), (OV), and (AC) are:

- $(C) = 1$ from the location of the bit 7 = 1, which means an error condition because the result cannot be represented in the unsigned scale (from 0_{10} to 255_{10}), i.e., in the 8-bit representation (greater than 255), as shown below:

$(A) =$	$A6_{16}$		$=$	166_{10}	
$(C) =$	1_{16}	$+$	$=$	1_{10}	$+$
$(21_{16}) =$	$7D_{16}$		$=$	125_{10}	
	124_{16}	$= 1.16^2 + 2.16^1 + 4.16^0$	$=$	292_{10}	$>255_{10}$

- $(AC) = 1$ of the position of the bit 3 = 1.
- $(F0) = 0$ same from the previous value. It is not affected by this instruction ($=0$).
- $(RS1)$ e $(RS0)$: they are the same from the previous values. They are not affected by this instruction ($=11_2$).
- $(OV) = 1$ of the position of the bit 7 or-exclusive (go 1 of the position of the bit 6) = 1 or-exclusive 1 = 0. It indicates that the result of this operation can be

stored on the signed scale, ranging from -128 to $+127$, and thus there is no error condition to represent this value on the signed scale.

The $A6_h$ value on the signed scale is considered as a negative number because its most significant bit is equal to 1. To determine this value on the signed scale, simply determine its two's complement $[(A)_{C2}] = 5A_h$.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
$(A)=A6_h$	1	0	1	0	0	1	1	0	
$(A)_{C1}=[A6_h]_{C1}$	0	1	0	1	1	0	0	1	
$(A)_{C2}=[A6_h]_{C2}=[A6_h]_{C1}+1$	0	1	0	1	1	0	1	0	
									$= -5A_h = -(5 \cdot 16^1 + 10 \cdot 16^0) = -90_{10}$

The value of the $(21_h) = 7D_h$ on the signed scale is a positive number because its most significant bit is zero, and its corresponding value is equal to 125 in decimal base. Therefore, the sum of these values on the signaled scale is the following:

$(A) =$	A6h		=	-90_{10}	
$(C) =$	1 _h	+	=	1_{10}	+
$(21_h) =$	7D _h		=	125_{10}	
	$^{124}_h$	$= 2 \cdot 16^1 + 4 \cdot 16^0$	=	36_{10}	

Note that the result of this operation, which is the number 36_{10} , and therefore it be stored on the signed scale from -128 to $+127$.

- $(P) = 0$, because it reflects the number of 1 s of the content of the accumulator (A) register that is equal to $24h=00100100_2$, which is equal to 2, even number of 1 s, i.e., the result of this operation has even parity.

Consequently, the content of the (PSW) register is equal to:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
$(PSW) =$	C	AC	F0	RS1	RS0	OV	-	P	
	1	1	0	1	1	0	0	0	$= D8_h$

2. Design a piece of software for the 8051 core in Assembly language that executes an OR-exclusive logic operation between the contents of the accumulator (A) register and a constant 33_h . The result must be stored in the content of the R5 register of the second register bank.

Answer

XRL A,#33h ; (A) \leftarrow (A) or-ex #33_h
MOV PSW,#08h ; (PSW) #08_h = # 00001000₂ (RS1)=0 e (RS0)=1 defines the second register
MOV R5,A ; bank of the 8051-core microcontroller
register of the second register bank. ; the contents of the accumulator (A) register is copied to the content of the R5

3. Regarding the following initial conditions, execute (run / process) the instructions theoretically, showing the registers and memory locations affected by the processing of such instructions:

(A) = E7_h; (PSW) = BC_h; (20_h) = 2F_h; (21_h) = 7E_h; (22_h) = 8C_h; (23_h) = 5B_h; (24_h) = 6A_h; (25_h) = 39_h; (26_h) = 4C_h; (R0) = 26_h; (R1) = 22_h; (R2) = E1_h; (R3) = 7D_h; (R4) = 8C_h; (R5) = 7B_h; (R6) = 6A_h; (R7) = 59_h.

(a) **ADDC A, 21h**

Answer The symbolic representation of the instruction is:

$$(A) \leftarrow (A) + (C) + (21h)$$

So:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
(A) = E7h = 231 ₁₀	11	11	11	10	10	11	11	1	
(C) = 1 _b	0	0	0	0	0	0	0	1	
(21h) = 7Eh = 126 ₁₀	0	1	1	1	1	1	1	0	
(A) \leftarrow (A) + (C) + (21h) = 358 ₁₀	10	1	1	0	0	1	1	0	= 66h = 102 ₁₀

In addition to the content of the accumulator (A) register changing from the value E7h to 66h, the content of the program status word register (PSW) changes to:

(PSW) =	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
	C	AC	F0	RS1	RS0	OV	-	P	
	1	1	1	1	1	0	0	0	= F8 _h

Considering:

- (C) = go 1 of the position of the bit 7 = 1. It means that the result cannot be represented regarding 8 bits (result = 358₁₀ > 255₁₀). Indicate an error condition because the result cannot be represented in the unsigned scale (from 0₁₀ to 255₁₀).

The final value in 9 bits can be obtained if the carry-bit flag is used to calculate the result. (C). $16^2+6 \cdot 16^1+6 \cdot 16^0=358_{10}=1.16^2+6.16^1+6 \cdot 16^0=358_{10}$;

- (AC) = go 1 of the position of the bit 3 = 1. If a decimal adjustment (DA A) operation is performed, the value 6 will be added to the least significant 4 bits to perform the conversion.
- (F0) = it is not affected by this instruction and continue with the same value = 1, because the initial value of the (PSW) register was equal to BC_h=10111100(indicated in red).
- (RS1,RS0) = it is not affected by this instruction and continues with the same value. RS1 = 1 and RS0 = 1, because the initial value of the (PSW) register was equal to BC_h=10111100(indicated in red).
- (OV) = (go 1 of the position of the bit 7) or-ex (go 1 of the position of the bit 7) = 1 or-ex 1 = 0. It means that there was no error condition in the numerical representation of the result within the range of -128 to +127.
- (P) = 0, because reflects the quantity of the numbers of 1 s of the content of the accumulator (A) register that is equal to 66_h=01100110₂, which is equal to 4, even number of 1 s, that is, the result of this operation has even parity.
- Besides, the zero flag (Z) is equal to 0 because the accumulator (A) register is different from zero.

(b) SUBB A, 20h

Answer The symbolic representation of the instruction is:

$$(A) \leftarrow (A) - (C) - (20h) = (A) + [(C) + (20h)]C_2 \Leftrightarrow (C) = \text{not } (C) \text{ e } (AC) = \text{not } (AC)$$

So, regarding (C) = 1 due to the initial value of the (PSW)=BC_h=10111100₂:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
$(20h) = 2F_h = 47_{10} =$	00	00	01	10	11	11	11	1
$(C) = 1_2 = 1_{10} =$								1
$(20h) + (C) = 30_h = 48_{10} =$	0	0	1	1	0	0	0	0
$[(R0) + (C)]C_1 = CF_h = 207_{10} = -29_{10} =$	01	01	00	10	11	11	11	1
$[(R0) + (C)]C_2 = D0_h = -48_{10}$	11	01	00	01	00	00	00	0
$(A) = E7_h = 231_{10} = -25_{10} =$	1	1	1	0	0	1	1	1
$(A) + [(R0) + (C)]C_2 = B7_h = 183_{10} = -73_{10} =$	11	0	1	1	0	1	1	1

- (C) and (AC) must be complemented (subtraction instruction). (C) = not (C) = not (1) = 0 \Rightarrow (C) = 0. This means that the result (18310) can be represented by 8 bits, regarding the unsigned scale (from 0_{10} to 255_{10}).
- (AC) = not (AC) = not (0) = 1 \Rightarrow (AC) = 1. If a decimal adjustment (DA A) operation is performed, the value 6 will be added to the least significant 4 bits to perform the conversion.
- (F0): it is not affected by this instruction = 1.
- (RS1,RS0) = it is not affected by this instruction and continues with the same value. RS1 = 1 and RS0 = 1 because the initial value of the (PSW) register was equal to $BC_h=10111100$ (indicated in red).
- (OV) = (go 1 of the position of the bit 7) or-ex (go 1 of the position of the bit 6) = 1 or-ex 1 = 0. It means that there was no error condition in the numerical representation of the result [$-73_{10} = (B7_h)_{C2}$] within the range from -128 to +127.
- (P) = 0 because it reflects the quantity of the numbers of 1 s of the content of the accumulator (A) register that is equal to $B7_h = 10110111_2$, which is equal to 6, an even number of 1 s, i.e.. the result of this operation has even parity.

So, the (PSW) is changed to:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
(PSW) =	C	AC	F0	RS1	RS0	OV	-	P	= 78 _h
	0	1	1	1	1	0	0	0	

- Besides, the zero flag (Z) is equal to 0 because the accumulator (A) register is different from zero.

(c) ORL A,26h

Answer The symbolic representation is $(A) \leftarrow (A) \text{ or } (26h)$.

So:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
(PSW) =	C	AC	F0	RS1	RS0	OV	-	P	= 39 _h
	1	1	1	1	1	1	0	1	

- The carry-bit flag (C), carry-bit flag (AC), and overflow-flag (OV) are not changed by logical operations. They remain with the previous value of (PSW).

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
$(A) = E7_h =$	1	1	1	0	0	1	1	1	
$(26_h) = 4C_h =$	0	0	1	0	1	1	0	0	
$(A) \leftarrow (A) \text{ or } (26_h) = EF_h =$	1	1	1	0	1	1	1	1	or

- $(P) = 1$ because it reflects the quantity of numbers of 1 s from the content of the accumulator (A) register that is equal to $EF_h = 11101111_2$, which is equal to 7, and an even number of 1 s, i.e., the result of this operation has odd parity.
- Besides, the zero flag (Z) is equal to 0 because the accumulator (A) register is different from zero.

(d) INC A

Answer The symbolic representation is $(A) \leftarrow (A) + 1$.

So:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
$(A) = E7_h = 231_{10} = -25_{10} =$	01	01	01	00	10	11	11	1	
#1 =	0	0	0	0	0	0	0	1	
$(A) = E8_h = 232_{10} = -24_{10} =$	01	1	1	0	1	0	0	0	+

Only the zero (Z) and parity (P) flags are affected. Therefore, $(Z) = 0$ (the result is different from zero) and $(P) = 0$ [even parity: quantity of numbers of 1 s is equal to 4 in the accumulator (A) register]. The rest of the flags are not affected.

(e) SWAP A

Answer The symbolic representation is $(A)_{7-4} \leftrightarrow (A)_{3-0}$.

So:

Before:

				$(A) = E7_h$		1	1	1	0	0	1	1	1
--	--	--	--	--------------	--	---	---	---	---	---	---	---	---

After:

				(A) = 7E _h	0	1	1	1	1	1	1	1	0
--	--	--	--	-----------------------	---	---	---	---	---	---	---	---	---

Only the parity (P) flag is affected. (P) = 0 [even parity: quantity of numbers of 1 s is equal to 6 in the accumulator (A) register]. The rest of the flags are not affected.

(f) MUL AB

Answer The symbolic representation is:

$$\{(A) \leftarrow [(A) * (B)]\}_{7-0}$$

$$\{(B) \leftarrow [(A) * (B)]\}_{15-8}$$

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
C	AC	F0	RS1	RS0	OV	-	P
(PSW) = BC _h	1	0	1	1	1	1	0

(A) = E7 _h	1	1	1	0	0	1	1	1
(B) = 02 _h	0	0	0	0	0	0	1	0

	0	0	0	0	0	0	0	0
	1	1	1	0	0	1	1	1
	0	0	0	0	0	0	0	+
	0	0	0	0	0	0	0	+
	0	0	0	0	0	0	0	+
	0	0	0	0	0	0	+	
	0	0	0	0	0	0	+	
	0	0	0	0	0	0	+	
	0	0	0	0	0	0	+	

0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Content of (B) register

Content of (A) register

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
C	AC	F0	RS1	RS0	OV	-	P
(PSW) = 3D _h	0	0	1	1	1	1	0

(A) = CE _h =	1	1	0	0	1	1	1	0
(B) = 01 _h =	0	0	0	0	0	0	0	1

Only the zero (Z) and parity (P) flags are affected. Therefore, (Z) = 0 (the result is different from zero) and (P) = 0 [even parity: quantity of numbers of 1 s is equal to 4 in the accumulator (A) register]. The rest of the flags are not affected.

(g) DIV AB

Answer The symbolic representation is:

- (A) \leftarrow quotient of the result of (A)/(B)
 (B) \leftarrow rest of the result of (A/B)

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	C	AC	F0	RS1	RS0	OV	-	P
(PSW) = BC _h	1	0	1	1	1	1	0	0
(A) = E7 _h	1	1	1	0	0	1	1	1
(B) = 02 _h	0	0	0	0	0	0	1	0
(A) = 73 _h	0	1	1	1	0	0	1	1
(B) = 01 _h	0	0	0	0	0	0	0	1
(PSW) = 39 _h	0	0	1	1	1	0	0	1

Only the zero (Z) and parity (P) flags are affected. Therefore, (Z) = 0 (result is different from zero) and (P) = 1 [even parity: quantity of numbers of 1 s is equal to 5 in the accumulator (A) register]. The rest of the flags are not affected.

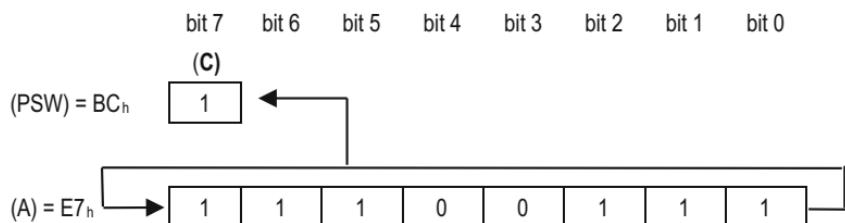
(h) RR A

Answer The symbolic representation is:

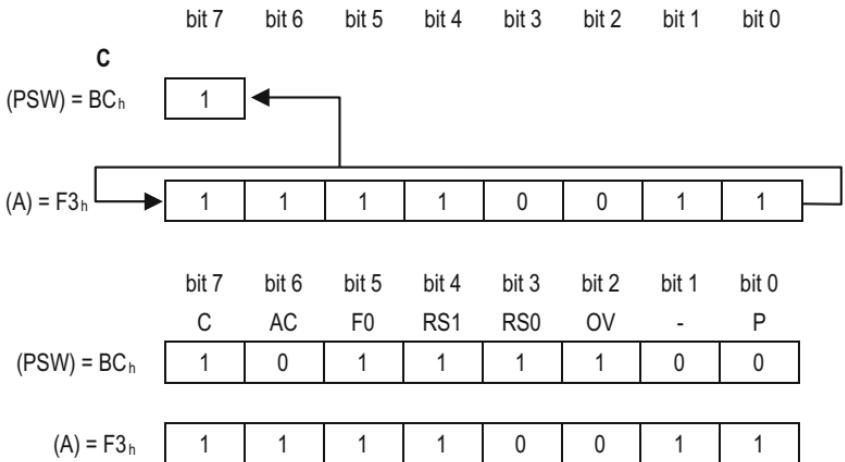
- $\{(A_n) \leftarrow (A_{n+1}) \text{ for } n = 0 \text{ to } 6$
 $\{(A_7) \leftarrow (A_0) \text{ and } (C) \leftarrow (A_0)$

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	c	ac	F0	RS1	RS0	ov	-	p
(PSW) = BC _h	1	0	1	1	1	1	0	0
(A) = E7 _h	1	1	1	0	0	1	1	1

Before:



After:

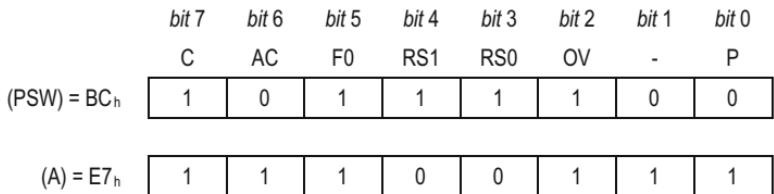


Only the carry-bit (C) and parity (P) flags are affected. (C) = 1 and (P) = 0 [even parity: quantity of numbers of 1 s is equal to 6 in the accumulator (A) register]. The rest of the flags are not affected.

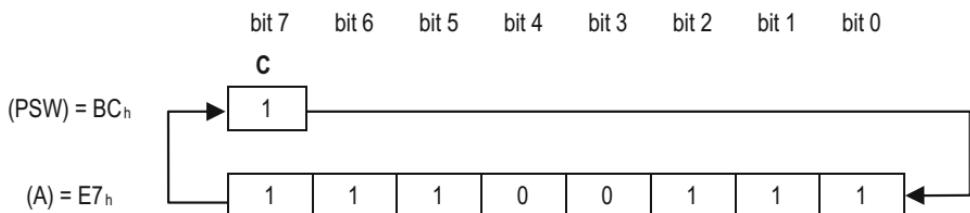
(i) RLC A

Answer The symbolic representation is:

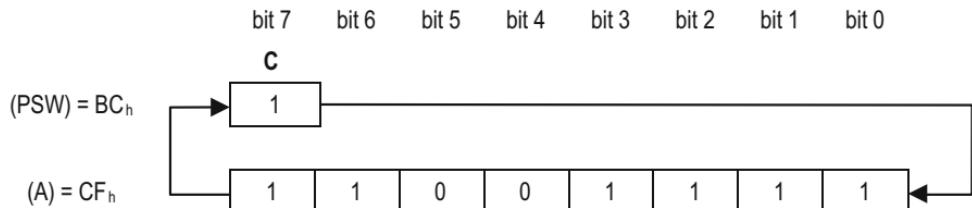
$$\{(A_{n+1}) \leftarrow (A_n) \text{ for } n = 0 \text{ to } 6\}$$
$$\{(A_0) \leftarrow (C) \text{ and } (C) \leftarrow (A_7)\}$$



Before:



After:



	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
	C	AC	F0	RS1	RS0	OV	-	P
(PSW) = BC _h	1	0	1	1	1	1	0	0
(A) = CF _h	1	1	0	0	1	1	1	1

Only the carry-bit (C) and parity (P) flags are affected. (C) = 1 and (P) = 0 [even parity: quantity of numbers of 1 s is equal to 6 in the accumulator (A) register]. The rest of the flags are not affected.

4. Considering the initial conditions from Exercise 1, calculate the contents of the program counter register (PC) for the following instructions:

Address	Mnemonic	Argument (s)	(PC) = ?
0142h	MOV	23h, @R1	
0208h	LJMP	30h	
0319h	DJNZ	R0, 0FAh	

Answer See Appendix A and verify how many bytes (column byte) each instruction occupied in the memory.

Address	Mnemonic	Argument(s)	(PC) = ?
0142h	MOV	23h, R1	(2 bytes) 0144h
0208h	LJMP	30h	(3 bytes) 020Bh
0319h	DJNZ	R0, 0FAh	(2 bytes) 031Bh

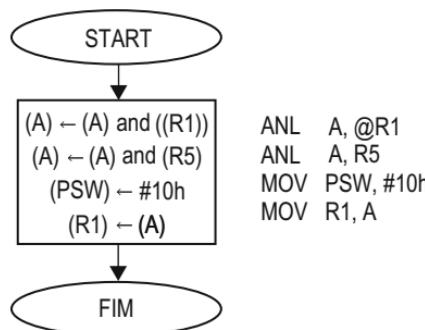
5. Implement a program (software) in Assembly language that adds the content of the accumulator (A) register and constant ACh and stores the result in the content of the memory location whose address is 4Bh.

Answer

```
Start: ADD A, #0ACh  
       MOV 4Bh, A  
       END
```

6. Implement a program (software) for the 8051 core in Assembly language that executes a logical AND operation of the contents of accumulator (A) register, the memory location whose address is given by the contents of R1 register and R5 register. The result must be stored in the content of the R1 register of the third register bank.

Answer



```
Start:     ANL    A, @R1  
           ANL    A, R5  
           MOV    PSW, #10h ; set the third register bank: 000100002  
           MOV    R1, A  
           END
```

3.6 Proposed Exercises

- 2.15.1. Regarding the following initial conditions, execute (run/process) the instructions theoretically, showing the registers and memory positions that are affected by the processing of such instructions and justify the result obtained considering the scales from 0 to 255 (unsigned scale) and from -128 to +127 (signed scale):

(A) = C7h; (PSW) = 8Ah; (PC) = 1000h; (20h) = 3Fh; (21h) = 6Eh;
(22h) = 7Ch; (23h) = CAh; (24h) = 9Bh; (25h) = 45h; (26h) = 81h;
(R0) = 77h; (R1) = 2Fh; (R2) = 1Ch; (R3) = 3Dh; (R4) = E2h;
(R5) = 21h; (R6) = FAh; (R7) = 21h; (3Fh) = 8Fh

I- ADDC A,@R0; II- SUBB A,R7; III- ADD A, R3; IV- SUBB A, 20h.

- 2.15.2. Design a piece of software for the 8051 core in Assembly language that executes an AND logic operation between the contents of the accumulator (A) register and the content of the R4 register. After, the result must be added to the content of the memory location whose address is given by the content of the R1 register. The result must be stored in the content of the memory location whose address is 33h.
- 2.15.3. Design a piece of software for the 8051 core in Assembly language that executes an addition operation between the contents of the memory location whose address is given by the content of the R0 register and the content of the memory location whose address is 7Ah. After, the result must be subtracted to the content of the R4 register. Later, the result must be rotated 2 bits to the right. The result must be stored in the content of the R3 register of the penultimate register bank.
- 2.15.4. Design a piece of software for the 8051 core in Assembly language that executes an addition operation between the contents of the R2 register and the content of the memory location whose address is 2Dh. After, the result must be done an OR logic operation with the content of the R4 register of the last register bank. Later, the result must be divided by two. The result must be stored in the content of the R1 register of the second register bank.

References

1. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
2. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
3. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
4. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
5. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
6. Atmel Corporation (1997) AT89 Series Hardware Description (0499B-B), Dec 1997
7. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001

8. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D–MICRO), June 2008
9. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
10. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

Flowchart and Assembly Programming

4.1 Introduction

This chapter is probably the most important of this book because it teaches and demystifies the main concepts of the Assembly programming and how we must proceed by using a systematic and straightforward way (really simple) to design software packages of any microprocessor/microcontroller of a computer system. The software is included in all electronics that use microprocessors or microcontrollers, such as electronics for personal use (calculators, digital clocks, cell phones, music players, tablets, etc.), residential (televisions, radios, refrigerators, microwave ovens, alarms, radios, etc.), car electronics (electronic injection, windscreen wiper, anti-lock braking system (ABS), autopilot, etc.), medical electronics (electronics of the x-ray, electrocardiogram, computerized tomography, magnetic resonance, etc.), commercial automation (cash registers, credit/debit card terminals, point terminals, data collectors, telephone exchanges, etc.), industrial (programmable logic controllers (PLC), electronic machine controllers, etc.), entertainment (gaming equipment, music players, etc.), and aerospace (satellites, spacecraft), i.e., the vast majority of the electronics we continually use [1–10].

It is easy to learn the secrets of these wonderful machines by practicing the programming techniques presented in this book. Besides, with this knowledge, it is possible to design your own electronics regarding this fantastic area of knowledge of electronic engineering. It is worth remembering that “what limits the design of an electronic product is your imagination” [1–10].

When you learn a low-level programming language (Assembly), you are automatically capable of learning any other programming language (from another manufacturer of microprocessors or microcontrollers, medium level such as the C language or high level such as Java, Basic, etc.) [1–10].

It is important to highlight that the software designer must know in detail the Assembly language of the microprocessor and microcontroller if he is using a medium- or high-level language. This is necessary because during the development of the software, it is often necessary to check the Assembly code generated by the

language used (medium/high level) to understand what is happening when an error condition occurs in the project by using the available computational tools of the language used, which must be solved (integration of the software with the hardware, test of the software in the hardware, emulation process, etc.). Therefore, it is fundamental that the software designer study the Assembly language of the microprocessor and microcontroller in detail before beginning the software development with these other programming languages (medium/high level) [1–10].

4.2 General Features of the Assembly Language

This chapter fundamentally explains step-by-step the design of sequential software packages and with loop by using the Assembly programming language of the 8051 core microcontroller family, due to its extensive range of manufacturers and users around the world. Once you learn and practice the programming techniques explained in this book, you can apply them to other programming languages (low, medium, or high level) [1–10].

There are many programming languages that can be used to program the computer systems by using microprocessors or microcontrollers, such as Assembly (low level), the structured C programming language (medium level), PL/M (high level), and unstructured and simple programming languages (Basic, Java, etc.) [1–10].

The programming language is quite critical and involves specialized technical knowledge. For instance, consider the cardiac pacemaker for humans. The main specifications of this electronic equipment are ideally it must be compact (the program memory and program must be as small as possible) to reduce the sensation of discomfort to the patient, it must have an infinite lifetime (with an extremely small electrical energy consumption to avoid the replacement of the battery and consequently the need for a new surgery), and it must generate electric pulses to help the heart works. In this context, it is important to highlight that the Assembly programming language is intrinsically the most capable of generating the small object code in comparison with the other languages (C, Java, Basic, etc.). Obviously, a pacemaker can be implemented with a high-level or medium-level programming language, but if the designer does not consider this information, it can generate an expensive electronic equipment with low penetration in the market and not achieve the fundamental objectives of this project, i.e., being cheap, compact, and with low power consumption. In each project, the designers must evaluate if the Assembly programming language is the most adequate to reach the desired specifications [1–10].

The characteristic of the Assembly programming language is to present the same set of instructions that is defined by the manufacturer of the microprocessor/microcontrollers. Consequently, it allows a full control of their internal registers and memory positions and a greater optimization capacity of the software (program memory reduced) than the other programming languages. Besides, whenever there are limitations of program memory and processing speed, it is recommended to use the low-level programming language, i.e., the Assembly language [1–10].

Regarding the characteristics of microprocessors and microcontrollers, it is very common to use the Assembly language to design computer systems, aiming at low cost and with great autonomy of the consumption power. In contrast, the only disadvantage of using the Assembly language in relation to the other ones is to require from the software designers a detailed prior knowledge of the internal architecture and functionality of the hardware of microprocessors/microcontrollers considered in the design. This previous knowledge differentiates a technical programmer in electronic engineering (electronic technicians and engineers) from a programmer that only knows how to design software regarding a specific hardware [1–10].

Remember that the fundamental goal to design the hardware and software of computer systems (electronics) is always to achieve the desired specifications and objectives presenting low cost, low power, high reliability, being compact, etc. The vast majority of engineers know about these features to design electronics, but to implement them regarding the concepts described in this book is an art for a few, since it involves well-trained professionals. Certainly, the high-technology companies that develop electronics know about these secrets and the need for these professionals [1–10].

The key features of the medium-level language (C language) is to present the characteristics of a low-level language (bit-by-bit manipulation of internal registers and memory locations, etc.) as well as the features of high-level languages (simple commands that do not require to know the hardware of the microprocessor or microcontroller: if and while instructions, etc.). It is currently one of the most used programming languages to design electronics that use microprocessors and microcontrollers, which do not involve rigorous engineering specifications or high production volume (one penny saved on electronics with high volume means big profits) [1–10].

The main feature of a high-level language is to be much closer to the language of the humans. It is descriptive regarding the instructions as they are described through words or word abbreviations written in English. Besides, it is very different from the instructions in Assembly and barely allows access to the microcontroller hardware. Each instruction of a high-level language is depicted by many instructions of a low-level language (Assembly) [1–10].

The programming with a high-level language is much easier than programming by using a low- or medium-level language. Larger software packages with lower processing speeds are generated when we use a high-level language. Another disadvantage of the high-level language, and often interpreted as an advantage, is not to require that the programmer have the complete and detailed knowledge of the device to be programmed but allowing another professional who does not belong to the technical engineering area to carry out simple application programs [1–10].

After editing, simulating, and compiling a piece of software, the machine code (object code) is generated. It is composed of bytes that represent the instructions which the microprocessor of the microcontroller is capable of performing. Instructions are commands that these integrated circuits can execute. Whenever a software is implemented in a particular language, compilation is required to

transform it into machine language (machine code or object code). The machine language software, after being simulated through a simulator (RIDE, AVSIM51, Pinnacle, etc.), can be recorded in the program memory of the hardware of the product so that we can experimentally test it to verify if the intended objectives have been achieved [1–10].

The success of programming in Assembly performed by the programmer is directly related to his previous knowledge of the hardware in which the product is being developed. Whenever a software designer develops a software package regarding a new hardware, he must first study the hardware of the microcontroller, and only afterwards he must implement the software. Therefore, this book aims to teach you how to implement software packages of 8051 core microcontrollers in Assembly; however, if you use another microcontroller, first you must study its hardware and only then you must implement the desired software [1–10].

4.3 Methodology to Implement Software Packages in Assembly

It is very important for a programmer, before implementing a software package, to follow a certain methodology. The technical prerequisite for developing software for a computer system implemented with a microcontroller is initially to know its internal architecture (hardware) and set of instructions. Besides, it is necessary to carry out the feasibility analysis of the project, which is determined by the clear understanding of the customer's specifications (customer wishes, product and system characteristics, operating mode, functioning conditions, cost, compactness, technology, etc.). It is also important to develop different solution strategies for the project, which can be done by using data flow diagram (DFD), block diagrams, or drawings [10].

Regarding the feasibility phase aiming to know the possibility to implement the computer system in the customer, it is necessary to ask some basic questions, such as the following: Is there a predetermined technology? What will be the final cost of the product (Will the customer pay for these electronics??) How many products will be manufactured (Is the factory able to produce these quantities??) How will the product be delivered? Are there suppliers of parts in our country? What components should be imported? Is there any competition?

Additionally, to choose the best solution, it is usually important to involve the customer in all aspects of the product. In this phase of the feasibility analysis, it is important to have a team of specialists from different areas of the company involved in the product (product, process, quality, purchasing, sales, marketing, etc.). The result of a feasibility analysis generates the final cost of the product, suppliers, technology, sales expectations, billing, investments, etc., or simply the non-realization of the project [10].

Once the project has been approved to be developed, after the feasibility analysis process has been completed, its implementation can begin with the elaboration of general and detailed block diagrams, in which the objectives of the hardware and

software of the product must be specified in detail. After that, we must implement a general flowchart or block diagram of the product, which must describe all features and objectives of the product. Afterwards, each block that describes each feature and objective must be detailed by using another general flowchart or block diagram. This procedure must be repeated until the software of the product is defined. In this phase, we must only use the flowchart to write the software of the product. In this flowchart, we must only use the symbolic representation of the instructions of the programming language which was chosen to write the software (flowchart associated with the programming language), which describes in detail the strategy used to implement the objective of the product desired (specifications of the product). Posteriorly, we must only generate the source program considering the chosen programming language (file in ASCII code that contains the instructions). Subsequently, the source program must be compiled to generate the binary file that corresponds to the machine codes of the software (microprocessor language). Usually, each part of the software must be simulated by using the computational tools of the development system of the software project (virtual verification if the software is working properly regarding the microcontroller considered). Once this phase is finished, it does not mean that the software developed will perfectly work in the hardware due to several reasons, such as another input/output bit being used, hardware not working, wrong software, etc. Then, it must be tested in the hardware of the product in order to verify if it answers to the software designed [10].

Afterwards, all files that compose the software of the product must be linked (link phase) to generate the complete software of the product. Linking is the final stage of program compilation and has the function of grouping all program blocks into binary in an organized way, which is set by the programmer. After the linkage process, it is recommended to run a simulation of the entire program before the test in the final product to verify if its operation meets the desired specifications. In the case of any errors, the software must be corrected [10].

Finally, we must record the complete software in the program memory of the microcontroller of the product in order to verify if all desired functionalities are met, according to the specifications of the product. In this phase, it is important to emulate the software by using the hardware developed, which consists of performing the physical simulation of the developed software aiming to correct the errors of logic or possible failures of any component that are used in the own hardware of the product. This process requires an emulation system [10].

After the emulation, it is necessary to do workbench tests to verify the functioning of the final product and then the tests of long duration, or reliability, in the conditions of stress aiming at the homologation of the product in relation to the specifications [10].

4.4 Development of Sequential Software in Assembly

The ideal technique to implement any software, which may be written in any programming language, is the flowchart. It is important to highlight that a programmer may not implement the flowchart of the software, but usually the cycle time of the development of the software can increase exponentially for the following reasons [10]:

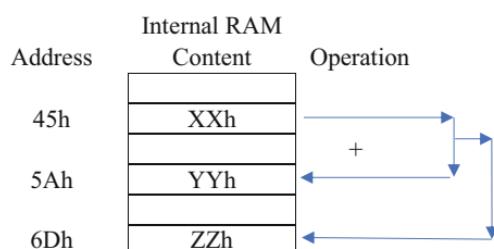
- The programmer does not have condition to see clearly what strategies were used to meet the specific objectives/specifications of the product. This occurs because the flowchart was not implemented (written).
- The flowchart helps the programmers write the strategies used to meet the specifications of the product.
- The flowchart helps the programmers think about how to implement the objectives to be reached by the product.
- The flowchart helps the programmer find the errors of a piece of software quickly.
- The flowchart helps the programmer perform changes in a piece of software quickly.
- The flowchart helps the programmer verify all potential possibilities that must be considered to achieve an objective of the product.
- Other.

To illustrate, implement the flowchart and source program in Assembly for the 8051 microcontroller, which is capable of adding two bytes. These bytes are in the contents of the memory locations, whose addresses are 45h and 5Ah, respectively. The result of this arithmetic operation must be stored in the content of the memory location whose address is 6Dh [10].

First, you need to fully understand what is being asked for. Therefore, a sketch-shaped drawing, which is illustrated in Fig. 4.1, can be considered a powerful tool to do so.

Then, it is necessary to define the strategy to solve the problem. To do this, first you need to verify what instructions are capable of performing the addition operation in the instruction set of the 8051 microcontroller (Appendix A). We can observe that there are eight instructions that perform the addition operation (ADD and ADDC with different address modes), which can be used to solve the problem proposed. By analyzing the requested situation, we do not use the ADDC instruction because it also considers the carry-bit in the addition instruction, and, in this case, this

Fig. 4.1 A sketch-shaped drawing to help in understanding what is being requested



consideration is not requested from us. Therefore, we must only consider the four ADD statements with their different address modes (four possibilities of solution for this problem).

I: By Considering the Instruction ADD A, Rn: $(A) \leftarrow (A) + (R_n)$ [The content of the accumulator (A) register is copied with the result of the addition operation between the contents of the accumulator (A) and R0 registers.]

To solve the problem with this instruction, it is necessary to use two registers to perform an addition operation, i.e., the contents of the accumulator (A) register and the content of one of the registers from one of the selected banks [(R0) to (R7)]. Considering the first register bank (register bank 0), it is necessary to perform two operations of moving data from the memory to the accumulator and from the memory to one of the registers before executing the addition instruction ADD A, Rn. First, the content of the memory location whose address is 45h must be copied to the content of the accumulator (A) register, i.e., MOV A, 45h [$(A) \leftarrow (45h)$]. Then, we must copy the content of the memory location whose address is 5Ah to the content of the R0 register (it could be any register from the register bank 0: R0, ..., R7). In this case, we have chosen the R0 register), i.e., the instruction MOV R0, 5Ah [$(R_0) \leftarrow (5Ah)$]. Then the ADD A, R0 [$(A) \leftarrow (A) + (R_n)$] instruction must be executed. As the result of the addition operation is stored in the content of the accumulator (A) register, it must be also stored in the content of the memory data location whose address is 6Dh. Therefore, we have to use the instruction MOV 6Dh, A [$(6Dh) \leftarrow (A)$]. This program uses seven memory locations and wastes five clock cycles to solve this problem. The flowchart and source program are shown in Fig. 4.2.

II: By Using the Instruction ADD A, Direct: $(A) \leftarrow (A) + (\text{direct})$ [The content of the accumulator (A) register is copied with the result of the addition operation

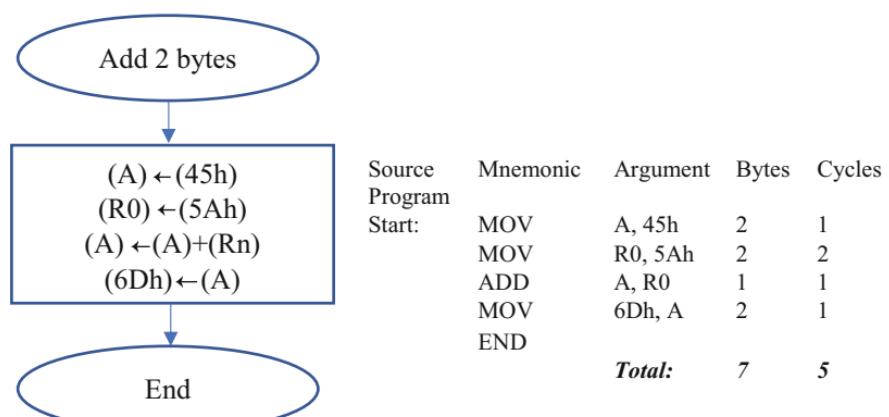


Fig. 4.2 The flowchart and source program of the addition program of two memory locations by using the ADD A, Rn instruction

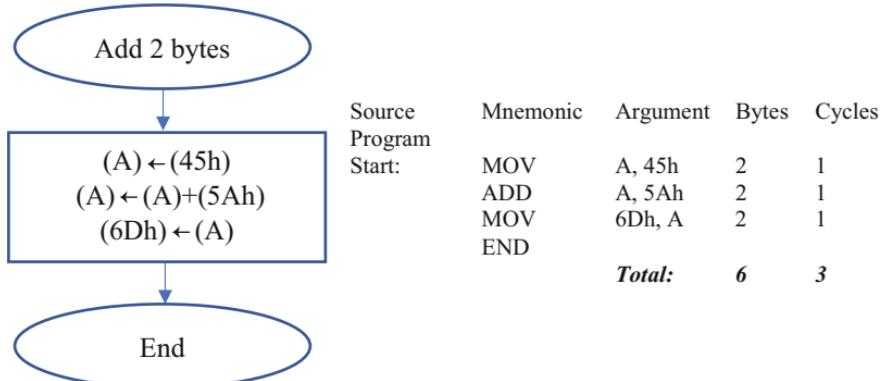


Fig. 4.3 The flowchart and source program for adding two memory locations using the ADD A, direct instruction

between the content of the accumulator (A) register and the content of the memory location whose address is given by the direct value.]

To resolve this problem with this instruction, you must first move the content of the memory location whose address is 45h to the content of the accumulator (A) register, i.e., MOV A, 45h [(A)←(45h)], as it was done for the previous solution. Afterwards, we must use the instruction ADD A, 5Ah [(A)←(A)+(5Ah)], setting the value 5Ah as direct [address of the memory location to be added to the content of the accumulator (A) register]. As the result is stored in the content of accumulator (A) register, it must be also stored in the content of the memory location whose address is 6Dh, i.e., MOV 6Dh, A [(6Dh)←(A)], exactly as it was previously done regarding the other solution. This program uses six memory locations and wastes three clock cycles to solve this problem. The flowchart and source program are shown in Fig. 4.3.

III: By Using the Instruction ADD A,@Ri: (A)←(A)+((Ri)) [The content of the accumulator (A) register is copied with the result of the addition operation between the content of the accumulator (A) register and the content of the memory location whose address is given by the content of the Ri (only R0 and R1) register.]

To solve this problem with this instruction, we must copy again the content of the memory location whose address is 45h to the content of the accumulator (A) register, i.e., MOV A, 45h [(A)←(45h)]. Then, we must initialize the content of the R0 register (the register R1 can also be used) with the value of the address of the memory location in which we desire to perform the addition operation that in this case is equal to 5Ah, i.e., MOV R0,#5Ah [(R0)←#5Ah]. Then, we must perform the addition instruction by using the ADD A,@R0 [(A)←(A)+((R0))]. As the result is stored in the content of the accumulator (A) register, it must be stored in the content of the memory location whose address is 6Dh, i.e., MOV 6Dh, A [(6Dh)←(A)]. This program uses seven

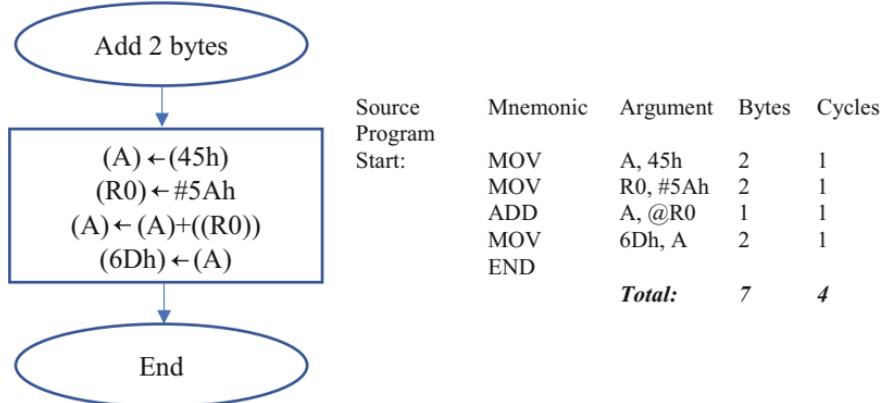


Fig. 4.4 The flowchart and source program of addition of two memory locations by using the ADD A, @Ri instruction

memory locations and wastes four clock cycles to solve this problem. The flowchart and source program are shown in Fig. 4.4.

IV: By Using the Instruction ADD A,#data: (A)←(A)+#data [The content of the accumulator (A) register is copied with the result of the addition operation between the content of the accumulator (A) register and the constant (indicated by #) date.]

In this case, there is no solution as it is not possible to know the value of the content of a memory location. Actually, in the content of a memory position, it is possible to store any value between 00h and FFh, i.e., a memory position is considered as a variable. Therefore, this instruction is not capable of producing a solution for this problem.

Regarding the three different ways to solve this problem, we can conclude that:

- Although there are four different address modes of the ADD instruction, there are only three feasible solutions.
- The best software for this problem is the one that uses the “direct” address mode (ADD A, direct) because this software uses a smaller number of instructions, six in this case (low cost), and wastes a smaller quantity of clock cycles, three in this case (processing speed faster), as indicated in Figs. 4.2, 4.3, and 4.4, columns “Bytes” and “Cycles” (data obtained of the Appendix A). This is defined as the optimal solution. The hardware/software designer must always present the best solution for the problem.
- This example must be used as reference by the software designer to implement other projects.

4.5 Development of Software with Loop in Assembly

Based on the concepts presented in Chap. 1, Sect. 1.12, this section describes the methodology for detailed implementation of the software with loop in Assembly [10].

So, consider that we must calculate the quantity of numbers which are smaller than 38h of a memory buffer (a memory range with the same sort of data, such as age, weight, height, etc.), whose initial and final addresses are 60h and 7Ah, respectively. This quantity must be stored in the content of the memory location whose address is 7Bh [10].

There are many ways to solve this problem. One of them is presented below, but it is one of the most optimized. If this software has been solved through a sequential program, we would have to use a lot of instructions and memory locations to do this because we have $7Ah - 60h + 1 = 1Bh = 27_{10}$ elements in the memory buffer to be analyzed. Thus, it is not recommended to solve this problem by the sequential method. Therefore, the best way to solve this problem is to use the flowchart of the software with loop, which was suggested in Chap. 1, Sect. 1.12. However, we must initially understand clearly what is being requested. Again, a rough draft drawing is used to illustrate what is being asked for, as shown in Fig. 4.5.

First, we must define the name of the software that in this case is “Qty. of numbers < value 38h.”

Initially, we have to define the initial conditions of the problem. To do this, we must define the control variables of the memory buffer. In this case, we are using the R0 register to store the initial value of the address of the memory buffer. This register is initialized with the value 60h. Besides, another register is necessary to control the memory buffer. This register must store the number of elements contained in the memory buffer. In this case, we are using the R1 register to do this, which is initialized with the value 1Bh ($7Ah - 60h + 1 = 27_{10}$). Additionally, as we must calculate the quantity of numbers which are smaller than 38h, we must have a variable, i.e., a counter to store the quantity of numbers which are smaller than 38h from this memory buffer. This counter must be initialized with an initial value equal to zero, and it must be incremented with one unit every time that an element of the memory buffer is smaller than 38h. As the quantity of the elements which are smaller than

Internal RAM		
	Address	Content
Initial address	60h	XXh
	:	:
	:	:
Final address	7Ah	YYh
	7Bh	Qty. of numbers < 38h

Fig. 4.5 A draft drawing to help us understand what is being requested for this problem

38h must be stored in the content of the memory location whose address is 7Bh, for convenience, this memory location is adopted to be the counter.

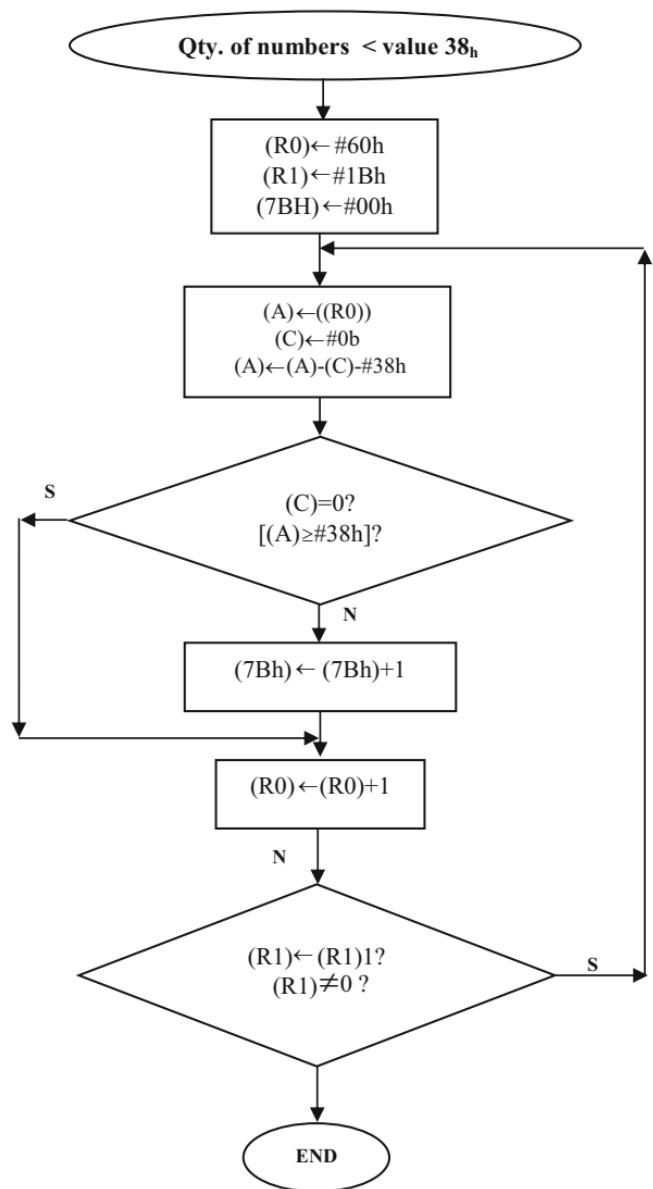
After that, we must define the block of the processing, testing, and decision-making to identify the quantity of elements which are smaller than 38h. This step defines the strategy to solve the problem. A possible approach is to analyze the content of each memory position of the memory buffer and compare it with the value 38h. A comparison operation can be done through the subtraction operation SUBB A, #data $[(A) \leftarrow (A) - (C) - \#data]$, where data is equal to 38h, and posteriorly we must test the carry-bit flag. The immediate addressing mode (#data) was used because we know the number that we have to subtract from the accumulator (A) register. If we used another addressing mode, the software would increase its size (this is not recommended). Before performing the subtraction operation, we must copy the content of the first memory location of the memory buffer to the accumulator (A) register, through the instruction MOV A, @R0 $[(A) \leftarrow ((R0))]$. In this case, we are using the R0 register because it contains the first address of the memory buffer. Besides, the content of the carry-bit flag (C) must be reset ($=0$) so that it does not affect the subtraction operation of the value 38h. Then, we can analyze the carry-bit flag (C) that is changed by the ULA after running the subtraction instruction. If the carry-bit flag is equal to zero $[(C)=0]$, it means that the content of the accumulator (A) register that contains the first content of the memory location of the memory buffer is greater than or equal to the value 38h, and therefore we do not have to add one unit to the counter of number smaller than 38h (content of the memory location 7Bh). If the carry-bit flag is equal to zero $[(C)=1]$, it means that the content of the accumulator (A) register that contains the first content of the memory location of the memory buffer is smaller than the value 38h, and therefore we must add one unit to the counter of number smaller than 38h (content of the memory location 7Bh). This is done by using the instruction INC 7Bh $[(7Bh) \leftarrow (7Bh)+1]$. Consequently, we must use the instruction JNC address {if $(C)=0$ $[(A) \geq \#38h$ (value equal to 38h)] then $(PC) \leftarrow$ "address," where "address" is the address of the instruction after the instruction INC 7Bh, i.e., this instruction aims to jump the instruction INC 7Bh because the number analyzed is higher than or equal to the value 38h} after the subtraction instruction. Therefore, as a tip, always when we must analyze the quantity of numbers which are smaller than a specific value, we have to test if the number analyzed is higher than or equal to the specific value to jump the counter (this strategy can be used for other tests, i.e., the test must be always contrary to what is being requested). Therefore, this approach defines the strategy to find the number of elements of the memory buffer that are smaller than the value of 38h.

Finally, in this moment we have to perform the control of the loop, i.e., how many times we have to proceed with our strategy that was described before to scan all elements of the memory buffer. To do this, we must increment the content of the R0 register in order to point to the next address of the memory buffer to be analyzed, which in this case is the address 61h. This is done by using the instruction INC R0 $[(R0) \leftarrow (R0)+1]$. Posteriorly, we must decrease one unit from the content of the R1 register that contains the quantity of elements of the memory buffer and verify if it is different from zero. If it is different from zero, it means that we have to proceed regarding the same approach done with the first memory position of the memory

buffer. This is done by using the instruction $\text{DJNZ R1, address } [(R1) \leftarrow (R0)-1]$ and if $(R1)$ is different from zero, the $(\text{PC}) \leftarrow$ "address," where "address" is the address of the define the first instruction of the block responsible for performing the processing, testing, and decision-making to identify the quantity of elements which are smaller than $38h$, as described before. Finally, after $1Bh=27_{10}$ loops, the software ends, and the quantity of elements of the memory buffer is determined and stored in the content of the memory location whose address is $7Bh$.

Figure 4.6 illustrates the flowchart and the source program in Assembly that calculates the quantity of elements of the memory buffer that are smaller than $38h$.

Fig. 4.6 The flowchart of the software in Assembly that calculates the quantity of elements of the memory buffer which are smaller than $38h$



The source program in Assembly of the proposed problem is described below.

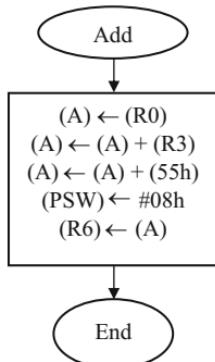
Address area	Mnemonic	Arguments	Comments
	MOV	R0, #60h	; Initialize the content of the R0 register with the ; initial address of the memory buffer
	MOV	R1, #1Bh	; Qty. of elements of the memory buffer. These two ; instructions are responsible for defining the ; memory buffer ($7Ah - 60h + 1 = 1Bh = 27_{10}$)
	MOV	7Bh, #00h	; Initialization of the counter [$(7Bh)$] which ; contains the quantity of elements of the memory ; buffer which are smaller than the value 38_h , ; which must initially be equal to zero
ADR2:	MOV	A, @R0	; Copy the content of the memory location which ; is given by the content of the R0 register (the ; data contained in the memory buffer) to the ; accumulator (A) register. Initially, the content ; of the memory location whose address is $60h$ is ; copied to the accumulator [$(60h)=XXh$]
	CLR	C	; Reset the carry-bit flag (C) so that it does ; not influence the result of the subtraction ; instruction
	SUBB	A, #38h	; Subtract the content of the accumulator (A) ; register of the carry-bit flag (C) and value $38h$
	JNC	ADR1	; If $(C)=0$ ($A \geq #38_h$) then $(PC) \leftarrow ADR1$ [it jumps ; to the instruction after the INC 7Bh, i.e., we must ; not add one unit to the content of the ; memory location whose address is equal to $7Bh$; (counter of elements of the memory buffer ; smaller than $38h$) because the element of the ; memory buffer is higher than or equal to the ; value $38h$]
	INC	7Bh	; But, if $(C)=1$ [$(A) < #38_h$] then we must add ; one unit to the content of the memory location ; whose address is equal to $7Bh$ (counter) because ; the element of the memory buffer is smaller ; than the ; value $38h$]
ADR1:	INC	R0	; This instruction adds one unit to the content ; of the R0 register in order to address the next ; memory location of the memory buffer to be ; analyzed (point to the next address of the other ; data to be analyzed)
	DJNZ	R1, ADR2	; This instruction decreases one unit from the ; content of the R1 register (qty. of elements of ; the memory buffer). If the quantity of elements ; of the memory buffer to be analyzed is different ; from zero, the software must jump to ADDR2 ; (it must proceed regarding the same approach ; performed for the previous element)
	END		; End

4.6 Exercises Solved

This section presents some exercises solved in order to teach the readers how to proceed to implement sequential software packages and with loop.

- I. Implement the flowchart and source program in Assembly regarding an 8051 core microcontroller which is able to add the contents of the R0 and R3 registers and the contents of the memory location whose address is 55h. The result must be stored in the content R6 register of the second register bank.

Solution Use the instructions set of the 8051 core in order to identify the instructions which are capable of performing the commands required.



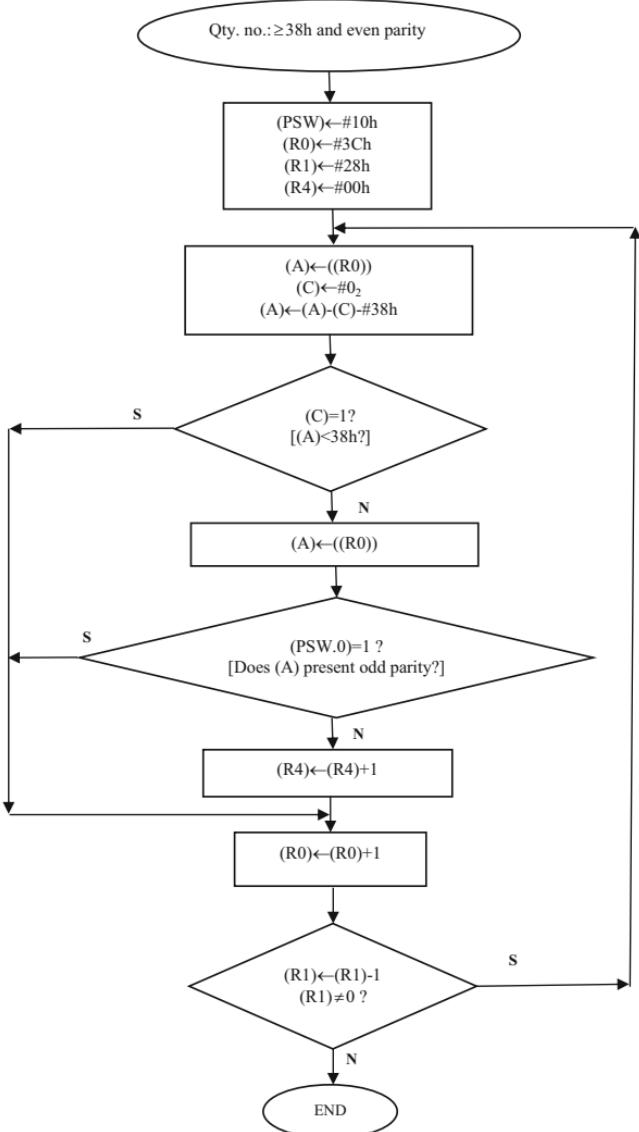
Address area	Mnemonic	Arguments	Comments
Start	MOV	A, R0	; Copy the content of the R0 register to the content of ; the ; accumulator (A) register
	ADD	A, R3	; Add the contents of the accumulator (A) and R3 ; registers, ; and the result is copied to the accumulator ; (A) register
	ADD	A, 55h	; Add the contents of the accumulator (A) and ; memory ; location whose address is 55h. The result is copied ; to the ; content of the accumulator (A) register
	MOV	PSW, #08h	; The RS1 and RS0 bits of the content of PSW are set ; to 0 ; and 1, respectively, i.e., they select the register bank ; 1 ; (second register bank)

(continued)

	MOV	R6, A	; Copy the content of the accumulator (A) register to ; the ; content of the R6 register
	END		; End

- II. Implement the flowchart and source program in Assembly for the microcontroller with the 8051 core which calculates the quantity of numbers higher than or equal to 38h and they present even parity of a memory buffer whose initial and final addresses are 3Ch and 63h, respectively. The quantity of elements with these characteristics must be stored in the content of the R4 register of the penultimate bank of registers (bank 2).

Solution One approach to solve this problem is to execute the subtraction instruction to compare the values of the memory buffer (one at a time) with the values 38h. After that, you must test the carry-bit flag (C). The carry-bit flag (C) is used to identify if the value of the memory buffer is higher than or equal to 38h or smaller than 38h. If the result of the subtraction operation is higher than or equal to 38h, the carry-bit flag is reset (=0); else (the result of the subtraction operation is smaller than 38h), the carry-bit flag (C) is set (=1). Besides, the parity-bit flag reflects the content of the accumulator (A) register. If the (PSW.0) is equal to zero, its means that the element of the buffer presents even parity; else (equal to 1), it means that the element of the buffer presents odd parity.



Address area	Mnemonic	Arguments	Comments
	MOV	PSW, #10h	; Set the penultimate register bank (bank 2). ; The RS1 and RS0 of the (PSW) must be equal ; to 1 and 0, respectively
	MOV	R0, #3Ch	; Initialize the content of the R0 register with the ; initial address of the memory buffer
	MOV	R1, #28h	; Qty. of elements of the memory buffer to be ; analyzed (final address subtracted from the first ; address, and added to one unit: ; 63h - 3Ch + 1 = 28h = 40 ₁₀).

(continued)

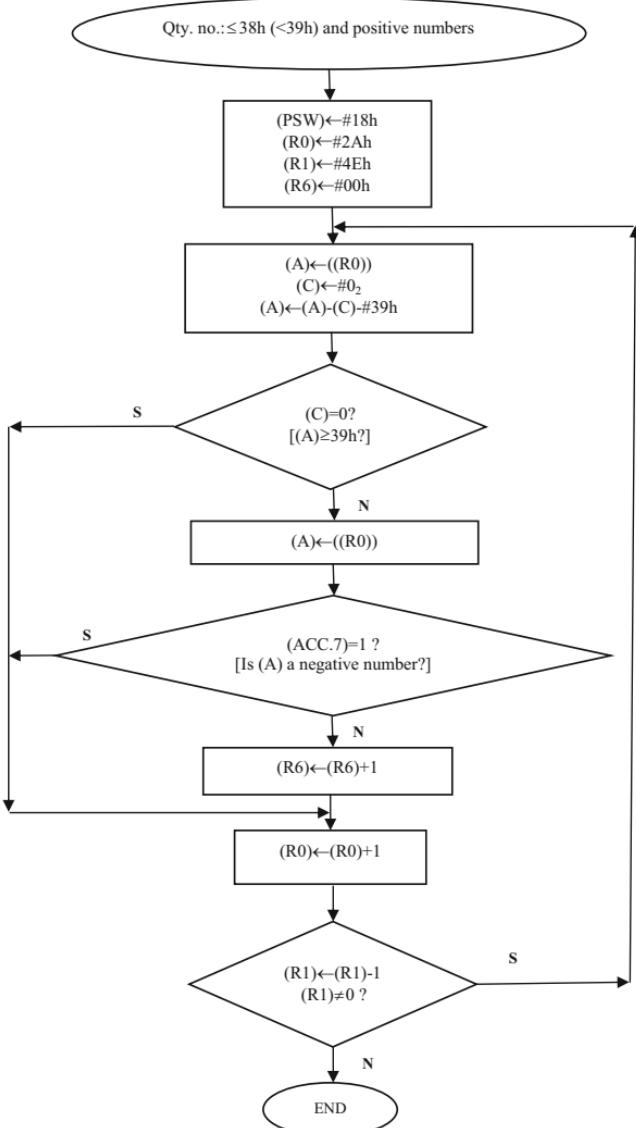
			; These two previous instructions are responsible ; for defining the memory buffer
	MOV	R4, #00	; Initialization of the counter [(R4)] which ; contains the quantity of elements of the memory ; buffer that are higher than or equal to 38h and ; present even parity. This counter must be initially ; with the zero value
ADR2	MOV	A, @R0	; Copy the content of the memory location whose ; address is given by the content of the R0 ; register (the data contained in the memory ; buffer) to the accumulator (A) register. Initially ; (A) \leftarrow (3Ch)
	CLR	C	; Reset the carry-bit flag (C) so that it does not ; influence the result of the subtraction ; instruction
	SUBB	A, #38h	; Subtract the content of the accumulator (A) ; register of the carry-bit flag (C) and value 38h
	JC	ADR1	; If (C)=1 ($A < \#38_h$) then (PC) \leftarrow ADR1 [it jumps ; to the instruction after the INC R4, i.e., we must ; not add one unit to the content of the ; R4 register (counter of elements of the memory ; buffer higher than or equal to 38h and that ; presents even parity) because the element ; of the memory buffer is smaller than the value 38h]
	MOV	A, @R0	; Copy the content of the memory location whose ; address is given by the content of the R0 ; register (the data contained in the memory ; buffer) to the accumulator (A) register again. ; This is done because the content of accumulator ; (A) register was changed by the subtraction ; instruction
	JB	PSW.0, ADR1	; If bit 0 of the (PSW) register (parity bit, P) ; is equal to 1 (odd parity), then (PC) \leftarrow ADR1 ; [jumps to the instruction after the INC R4, ; i.e., we must not add one unit to the content ; of the R4 register (counter of elements of the ; memory buffer higher than or equal to 38h and ; that presents even parity), because the element ; of the memory buffer does not present even ; parity]
	INC	R4	; If bit 0 of the (PSW) register is equal to zero (it ; presents even parity), then we must add one ; unit to the content of the R4 register (counter of ; elements of the memory buffer which are higher ; than or equal to 38h and that present even ; parity) because bit 0 of the (PSW), i.e., the parity ; bit of the element of the memory buffer is equal ; to 0 (it presents even parity)]
ADR1:	INC	R0	; This instruction adds one unit to the content ; of the R0 register in order to address the next

(continued)

			; memory location of the memory buffer to be ; analyzed (point to the next address of the other ; data to be analyzed)
	DJNZ	R1, ADR2	; This instruction decreases one unit from the ; content of the R1 register (responsible for ; storing the qty. of elements of the memory ; buffer to be analyzed). If this quantity to be ; analyzed is different from zero, this means that ; the software must jump to ADDR2 in order to ; repeat the same previous procedure for the next ; element of the memory buffer
	END		; End

III. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which calculates the quantity of numbers which are smaller than or equal to 38h and positive numbers (>0), regarding the signed scale of values, of a memory buffer whose initial and final addresses are 2Ah and 77h, respectively. The quantity of elements with these characteristics must be stored in the content of the R6 register of the last bank of registers (bank 3).

Solution One approach to solve this problem is to execute the subtraction instruction to compare the values of the memory buffer (one at a time) with the values 39h, because a number that is smaller than or equal to 38h is the same as saying that it is smaller than 39h. This approach is necessary to be performed because when we use the subtraction instruction to compare two values and (C) is reset ($=0$), it means that the result is higher or equal to 39h, and when the (C) is set ($=1$), it means that the result is smaller than 39h (smaller than or equal to 38h). After that, you must test the carry-bit flag (C). The carry-bit flag (C) is used to identify if the value of the memory buffer is higher than or equal to 39h or smaller than 39h (smaller than or equal to 38h). If the result of the subtraction operation is higher than or equal to 39h, the carry-bit flag is reset ($=0$); else (the result of the subtraction operation is smaller than 39h), the carry-bit flag (C) is set ($=1$). Besides, in order to test whether the number is positive (>0) or negative (<0), we have to test the sign-bit, which is defined as the most significant bit of the byte (bit 7). The accumulator (A) register is usually used to test if the number is positive or negative, as it is also addressed by bit. Therefore, before we test the most significant bit of the accumulator (A) register, we must copy the content of the data to be tested in the content of the accumulator (A). If the content of bit 7 of the accumulator (A) register (ACC.7) is equal to 0, its means that the number is positive, else (equal to 1), its means that it is a negative number.



Address area	Mnemonic	Arguments	Comments
	MOV	PSW, #18h	; Select the last register bank (bank 3). The ; RS1 and RS0 of the (PSW) must be equal to 1 ; and 1, respectively
	MOV	R0, #2Ah	; Initialize the content of the R0 register with the ; initial address of the memory buffer
	MOV	R1, #4Eh	; Qty. of elements of the memory buffer to be ; analyzed ($77h - 2Ah + 1 = 4Eh = 78_{10}$). These two ; previous instructions are responsible for ; defining the memory buffer

(continued)

	MOV	R6, #00	; Initialization of the counter [(R6)] which ; contains the quantity of elements of the memory ; buffer that are smaller than or equal to 38h and ; positive numbers, which must initially be ; equal to zero
ADR2	MOV	A, @R0	; Copy the content of the memory location which ; is given by the content of the R0 register (the ; data contained in the memory buffer) to the ; accumulator (A) register
	CLR	C	; Reset the carry-bit flag (C) so that it does not ; influence the result of the subtraction ; instruction
	SUBB	A, #39h	; Subtract the content of the accumulator (A) ; register of the carry-bit flag (C) and value 39h, ; due to a number which is smaller than or equal ; to 38h, which is the same as a number smaller ; than 39h. This must be done because we must ; identify whether a number is higher than or ; equal to 39h in order to add one unit to the ; content of the R6 register
	JNC	ADR1	; If (C)=0 [(A) \geq #39 _h] then (PC) \leftarrow ADR1 [it ; jumps to the instruction after the INC R6, i.e., ; we must not add one unit to the content of ; the R6 register (counter of elements of the ; memory buffer must be smaller than 39h ; (\leq 38h) ; and be positive numbers) because the ; element of the memory buffer is higher than or ; equal to the value 39h]
	MOV	A, @R0	; Copy the content of the memory location whose ; address is given by the content of the R0 register ; (the data contained in the memory buffer) to the ; accumulator (A) register again. This is done ; because the content of accumulator (A) register ; was changed by the previous subtraction ; instruction
	JB	PSW.7, ADR1	; If bit 7 of the (A) register (sign-bit) is ; equal to 1 (negative number), then ; (PC) \leftarrow ADR1 [jumps to the instruction after ; the INC R6, i.e. ,we must not add one unit ; to the content of the R6 register (counter of ; elements of the memory buffer must be smaller ; than or equal to 38h (<39h) and be positive ; numbers) because the element of the memory ; buffer is not a positive number]
	INC	R6	; If bit 7 of the accumulator (A) register is ; equal to 0 (positive number), then we must add ; one unit to the content of the R6 register ; (counter of elements of the memory buffer ; which are smaller than or equal to 38h and

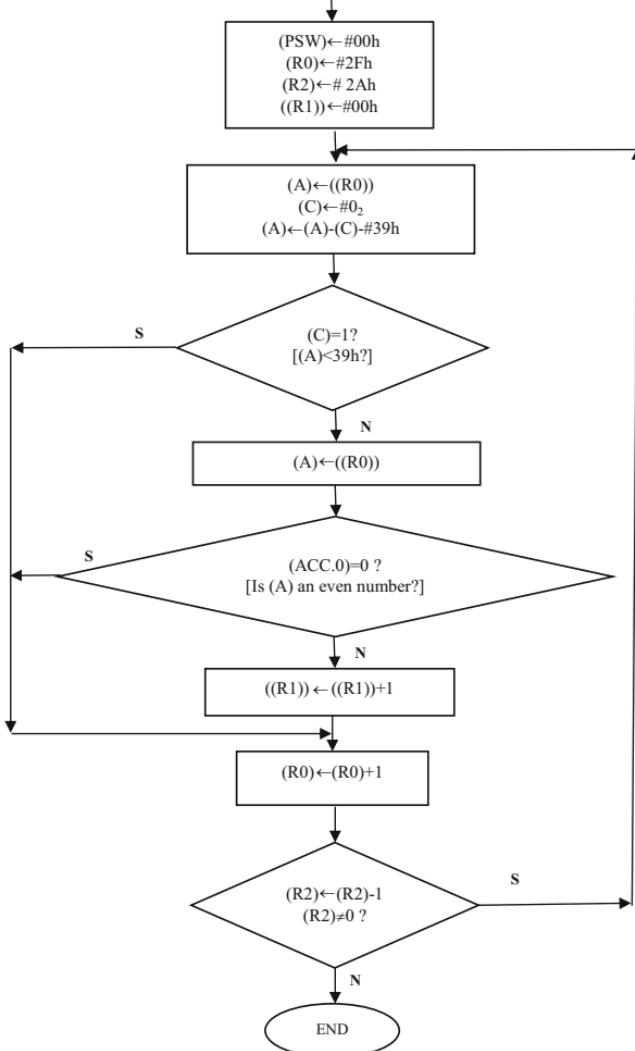
(continued)

			; positive numbers) because bit 7 of the ; element of the memory buffer is equal to 0]
ADRI:	INC	R0	; This instruction adds one unit to the content ; of the R0 register in order to address the next ; memory location of the memory buffer to be ; analyzed (point to the next address of the other ; data to be analyzed)
	DJNZ	R1, ADR2	; This instruction decreases one unit from the ; content of the R1 register (qty. of elements of ; the memory buffer to be analyzed). If the ; quantity of elements of the memory buffer to be ; analyzed is different from zero, the software ; must jump to ADDR2 in order to repeat the ; same procedure performed to the data ; considered before
	END		; End

IV. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which calculates the quantity of numbers which are higher than 38h and odd numbers, regarding the signed scale of values, of a memory buffer whose initial and final addresses are 2Fh and 58h, respectively. The quantity of elements with these characteristics must be stored in the content of the memory location whose address is given by the content of the R1 Register from the first register bank.

Solution One approach to solve this problem is to execute the subtraction instruction to compare the values of the memory buffer (one at a time) with the values 39h, as a number that is higher than 38h is the same as saying that it is higher than or equal to 39h. This strategy is necessary to be adopted because when we use the subtraction instruction to compare two values, if the (C) is reset (=0), it means that the result of the subtraction instruction is higher than or equal to 39h (>38h, which is the required), and when (C) is set (=1), it means that the result is smaller than 39h. After that, you must test the carry-bit flag (C). The carry-bit flag (C) is used to identify if the value of the memory buffer is higher than or equal to 39h (>38h) or smaller than 39h (smaller and equal to 38h). If the result of the subtraction operation is higher than or equal to 39h, the carry-bit flag is reset (=0); else (the result of the subtraction operation is smaller than 39h), the carry-bit flag (C) is set (=1). Besides, in order to test if the number is even or odd, we must test the least significant bit (bit 0) of the byte. The accumulator (A) register is usually used to test if the number is even or odd, as it is also addressed by bit. Therefore, before we test the least significant bit (bit 0) of the accumulator (A) register, we must copy the content of the data addressed by the content of the R0 register to be tested to the content of the accumulator (A). If the content of bit 0 of the accumulator (A) register (ACC.0) is equal to 0, its means that the number is even; else (equal to 1), it means that it is odd.

Qte. no.: >38h ($\geq 39h$) and odd numbers



Address area	Mnemonic	Arguments	Comments
	MOV	PSW, #00h	; Select the first register bank (bank 0). The ; RS1 and RS0 of the (PSW) must be equal to 0 ; and 0, respectively
	MOV	R0, #2Fh	; Initialize the content of the R0 register with the ; initial address of the memory buffer
	MOV	R2, #2Ah	; Qty. of elements of the memory buffer to be ; Analyzed ($58h - 2Fh + 1 = 2Ah = 42_{10}$). These two ; previous instructions are responsible for ; defining the memory buffer

(continued)

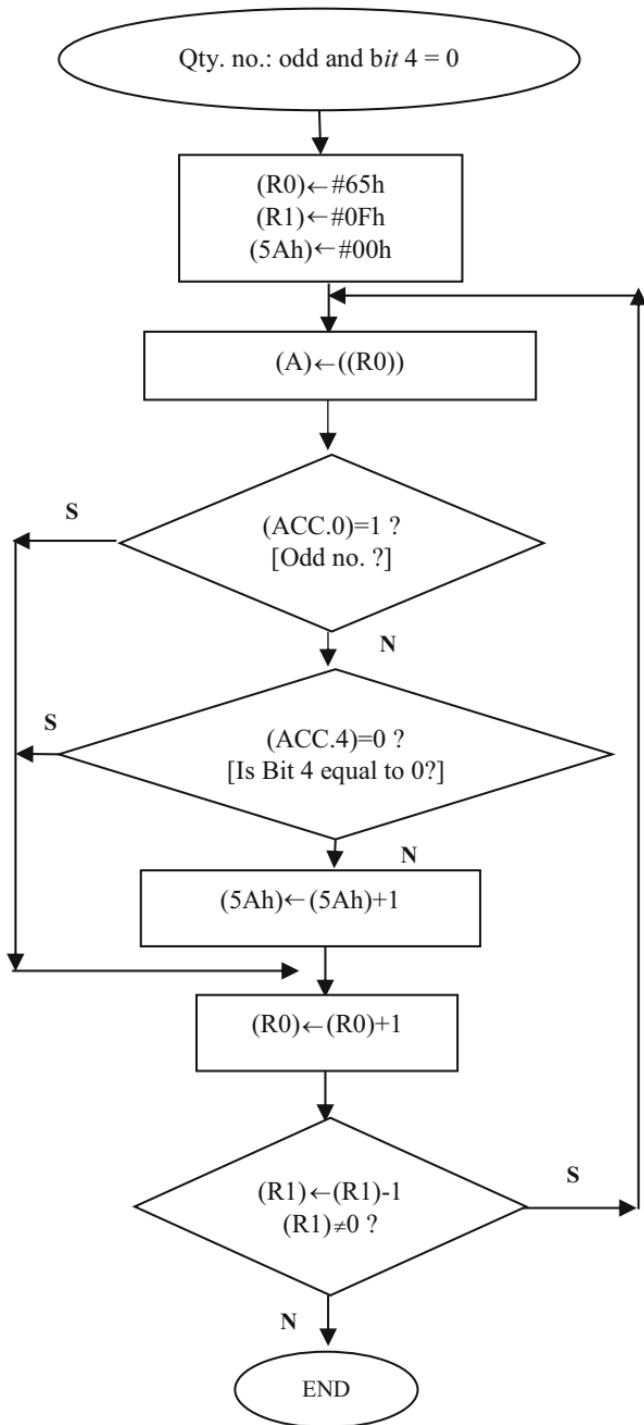
	MOV	@R1, #00	; Initialization of the counter [((R1))] which ; contains the quantity of elements of the memory ; buffer that are higher than 38h ($\geq 39h$) and ; odd numbers, which must initially be equal ; to zero
ADR2	MOV	A, @R0	; Copy the content of the memory location which ; is given by the content of the R0 register (the ; data contained in the memory buffer) to the ; accumulator (A) register
	CLR	C	; Reset the carry-bit flag (C) so that it does not ; influence the result of the subtraction ; instruction
	SUBB	A, #39h	; Subtract the content of the accumulator (A) ; register of the carry-bit flag (C) and value 39h, ; due to a number higher than 38h, which is the ; same as a number higher than or equal to 39h. ; This must be done because we must identify if ; a number is higher than 38h in order to add ; one unit to the content of the memory whose ; address is defined by the content of the R1 ; register
	JC	ADR1	; If $(C)=1$ [$(A)<\#39h$ ($\leq \#38h$)] then ; $(PC)\leftarrow ADR1$ [it jumps to the instruction after ; the INC @R1, i.e. we must not add one ; unit to the content of the memory whose address ; is defined by the content of the R1 register ; (counter of elements of the memory buffer ; higher than 38h ($\geq 39h$) and odd ; numbers) because the element of the memory ; buffer is smaller than the value 39h ($\leq 38h$)]
	MOV	A, @R0	; Copy the content of the memory location whose ; address is given by the content of the R0 register ; (the data contained in the memory buffer) to the ; accumulator (A) register again. This is done ; because the content of accumulator (A) register ; was changed by the previous subtraction ; instruction
	JNB	ACC.0, ADR1	; If bit 0 of the (A) register (least significant ; bit) is equal to 0 (even number), then ; $(PC)\leftarrow ADR1$ [jumps to the instruction after ; the INC @R1, i.e. we must not add one ; unit to the content of the memory which is ; addressed by the content of the R1 register ; (counter of elements of the memory buffer ; smaller than or equal to 38h ($<\#39h$) and that ; are odd numbers) because the element of ; the memory buffer is not a positive number]
	INC	@R1	; If bit 0 of the accumulator (A) register is ; equal to 1 (odd number), then we must add ; one unit to the content of the memory ; location whose address is given by the content

(continued)

			; of the R1 register (counter of elements of the ; memory buffer which are smaller than or equal ; to 38h and positive numbers) because ; bit 0 of the element of the memory buffer is ; equal to 0]
<i>ADRI:</i>	INC	R0	; This instruction adds one unit to the content ; of the memory location R0 register in order to ; address the next memory location of the ; memory buffer to be analyzed (point to the next ; address of the other data to be analyzed)
	DJNZ	R2, <i>ADR2</i>	; This instruction decreases one unit from the ; content of the R2 register (qty. of elements of ; the memory buffer to be analyzed). If the ; quantity of elements of the memory buffer to be ; analyzed is different from zero, the software ; must jump to ADDR2 in order to repeat the ; same procedure performed to the previous data ; considered
	END		; End

V. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which calculates the quantity of even numbers and presents bit 4 equal to 1 of a memory buffer whose initial and final addresses are 65h and 73h, respectively. The quantity of elements with these characteristics must be stored in the content of the memory location whose address is 5Ah.

Solution



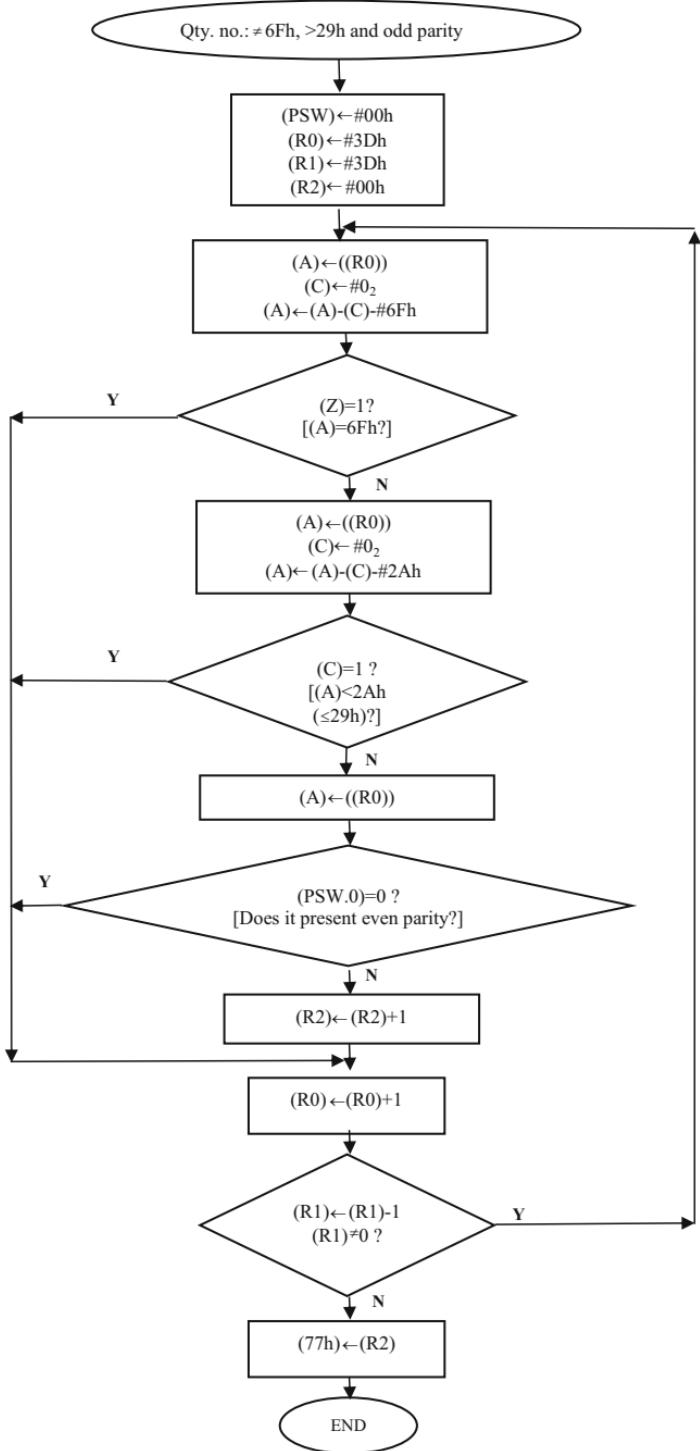
Address area	Mnemonic	Arguments	Comments
	MOV	R0, #65h	; Initialize the content of the R0 register with the ; initial address of the memory buffer
	MOV	R1, #0Fh	; Qty. of elements of the memory buffer. These ; two ; instructions are responsible for defining the ; memory buffer ($73h - 65h + 1 = 0Fh = 15_{10}$)
	MOV	5Ah, #00h	; Initialization of the counter [(5Ah)] which ; contains the quantity of elements of the memory ; buffer that are even numbers and present ; bit 4 equal to 1, which must be initially equal ; to zero
ADR2	MOV	A, @R0	; Copy the content of the memory location which ; is given by the content of the R0 register (the ; data contained in the memory buffer) to the ; accumulator (A) register
	JB	ACC.0, <i>ADR1</i>	; If bit 0 of the accumulator (A) register is ; equal to 1 (odd number), then $(PC) \leftarrow ADR1$; [it jumps to the instruction after the INC 5Ah, ; i.e. we must not add one unit to the ; content of the memory location whose address ; is equal to 5Ah (counter of elements of the ; memory buffer which is an even number and ; bit 4 equal to 1) because the element of the ; memory buffer is not an even number]
	JNB	ACC.4, <i>ADR1</i>	; If bit 4 of the accumulator (A) register is ; equal to 0, then $(PC) \leftarrow ADR1$ [it jumps to the ; instruction after the INC 5Ah, i.e., we must not ; add one unit to the content of the memory ; location whose address is equal to 5Ah (counter ; of elements of the memory buffer which are ; even numbers and bit 4 is equal to 1) ; because bit 4 of the element of the memory ; buffer is not equal to 1]
	INC	5Ah	; If bit 4 of the accumulator (A) register is not ; equal to 0, then we must add one unit to the ; content of the memory location whose address ; is equal to 5Ah (counter of elements of the ; memory buffer which are even numbers and the ; bit 4 equal to 1) because bit 4 of the element ; of the memory buffer is not equal to 1]

(continued)

<i>ADR1:</i>	INC	R0	; This instruction adds one unit to the content ; of the R0 register in order to address the next ; memory location of the memory buffer to be ; analyzed (point to the next address of the other ; data to be analyzed)
	DJNZ	R1, <i>ADR2</i>	; This instruction decreases one unit from the ; content of the R1 register (qty. of elements of ; the memory buffer). If the quantity of elements ; of the memory buffer to be analyzed is different ; from zero, the software must jump to ADDR2 (it ; must proceed regarding the same approach ; performed for the previous element)
	END		; End

VI. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which calculates the quantity of elements that are different from 6Fh, higher than 29h, and present odd parity of the memory buffer whose initial and final addresses are 3Dh and 79h, respectively. The quantity of elements with these characteristics must be stored in the content of the memory location whose address is 77h.

Solution



Address area	Mnemonic	Arguments	Comments
	MOV	PSW, #00h	; Select the first register bank (bank 0). The ; RS1 and RS0 of the (PSW) must be equal to 0 ; and 0, respectively
	MOV	R0, #3Dh	; Initialize the content of the R0 register with the ; initial address of the memory buffer
	MOV	R1, #3Dh	; Qty. of elements of the memory buffer to be ; analyzed ($79h - 3Dh + 1 = 3Dh = 61_{10}$). These two ; previous instructions are responsible for ; defining the memory buffer
	MOV	R2, #00	; Initialization of the counter [(R2)] which ; contains the quantity of elements of the memory ; buffer that are different from the value 6Fh, ; higher than 29h ($\geq 2Ah$), and present odd parity, ; which must initially be equal to zero. The R2 ; register was chosen to be a counter of elements ; of the memory buffer with these characteristics ; because the address 77h is being used to store ; data of the memory buffer
ADR2	MOV	A, @R0	; Copy the content of the memory location which ; is given by the content of the R0 register (the ; data contained in the memory buffer) to the ; accumulator (A) register
	CLR	C	; Reset the carry-bit flag (C) so that it does not ; influence the result of the subtraction ; instruction
	SUBB	A, #6Fh	; Subtract the content of the accumulator (A) ; register of the carry-bit flag (C) and value 6Fh
	JZ	ADR1	; If $(Z)=1$ [$(A)=#6Fh$], then $(PC) \leftarrow ADR1$ [it ; jumps to the instruction after the INC R2, i.e., ; we must not add one unit to the content of ; the R2 register (counter of elements of the ; memory buffer which are different from the ; value 6Fh, higher than 29h ($\geq 2Ah$) and present ; odd parity) because the element of the memory ; buffer is equal to the value 6Fh]
	MOV	A, @R0	; Copy the content of the memory location whose ; address is given by the content of the R0 register ; (the data contained in the memory buffer) to the ; accumulator (A) register again. This is done ; because the content of accumulator (A) register ; was changed by the previous subtraction ; instruction
	CLR	C	; Reset the carry-bit flag (C) so that it does not ; influence the result of the subtraction ; instruction
	SUBB	A, #2Ah	; Subtract the content of the accumulator (A) ; register of the carry-bit flag (C) and value 2Ah, ; due to a number which is higher than 29h and ; the same as a number which is higher than or

(continued)

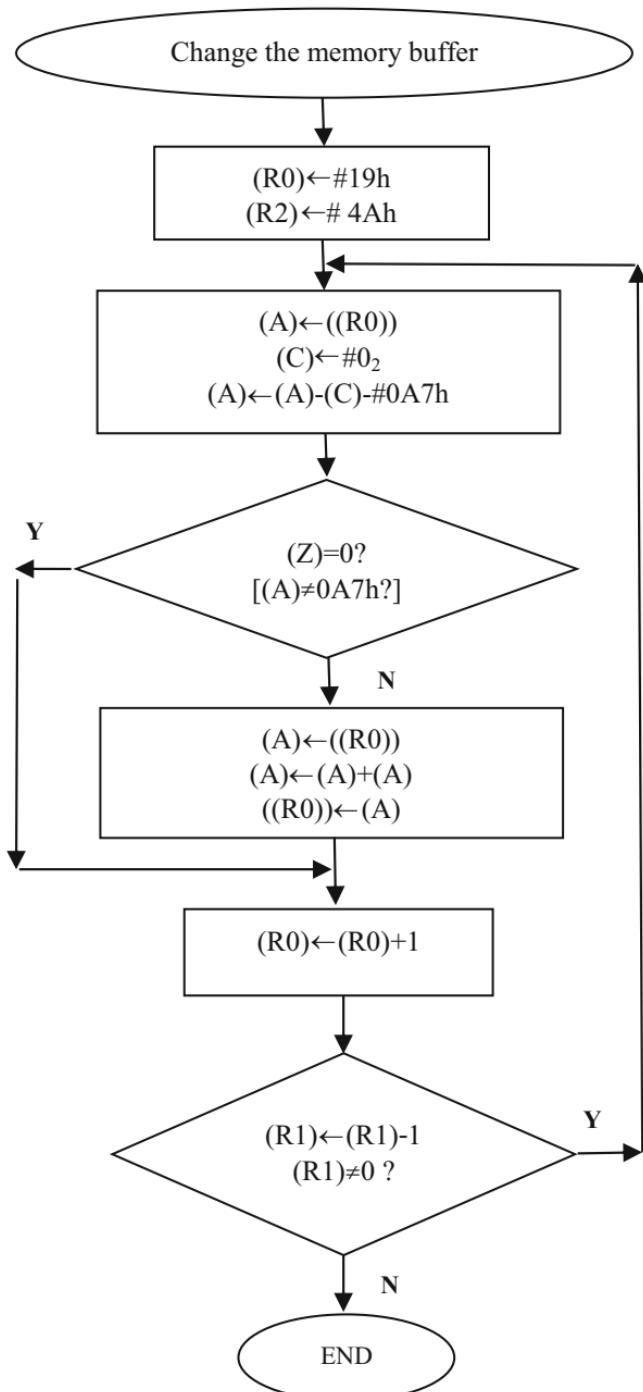
			; equal to 2Ah. This must be done because we ; must identify if a number is higher than 29h in ; order to add one unit to the content of the R2 ; register
	JC	<i>ADR1</i>	; If (C)=1 [(A)<#2Ah (\leq #29h)], then ; (PC) \leftarrow ADR1 [it jumps to the instruction after ; the INC R2, i.e., we must not add one ; unit to the content of the R2 register (counter of ; elements of the memory buffer which are ; different from the value 6 Fh, higher than 29h ; (\geq 2Ah), and present odd parity) because the ; element of the memory buffer is smaller than ; the value 2Ah (\leq 29h)]
	MOV	A, @R0	; Copy the content of the memory location whose ; address is given by the content of the R0 register ; (the data contained in the memory buffer) to the ; accumulator (A) register again. This is done ; because the content of accumulator (A) register ; was changed by the subtraction instruction ; before
	JNB	PSW.0, <i>ADR1</i>	; If bit 0 of the (PSW) register (parity bit) is ; equal to 0 (even number), then (PC) \leftarrow ADR1 ; [jumps to the instruction after the INC R2, i.e., ; we must not add one unit to the content of ; the R2 register (counter of elements of the ; memory buffer which are different from the ; value 6Fh, higher than 29h (\geq 2Ah) and present ; odd parity) because the element of the memory ; buffer does not present odd parity]
	INC	R2	; If bit 0 of the content of the PSW register is ; equal to 1 (odd parity), then we must add ; one unit to the content of the R2 register ; (counter of elements of the memory buffer ; which are different from the value 6Fh, higher ; than 29h (\geq 2Ah) and present odd parity)
<i>ADR1:</i>	INC	R0	; This instruction adds one unit to the content ; of the memory location R0 register in order to ; address the next memory location of the ; memory buffer to be analyzed (point to the next ; address of the other data to be analyzed)

(continued)

	DJNZ	R1, ADDR2	; This instruction decreases one unit from the ; content of the R1 register (qty. of elements of ; the memory buffer to be analyzed). If the ; quantity of elements of the memory buffer to be ; analyzed is different from zero, the software ; must jump to ADDR2 in order to repeat the ; same procedure performed to the previous data ; considered
	MOV	77h, R2	; Store the content of the R2 register in the ; content of the memory location given by the ; address 77h, as requested. The content of the ; address 77h was not used as a counter because ; it belongs to the memory buffer to be analyzed
	END		; End

VII. Implement the flowchart and source program in Assembly for the microcontroller with the 8051 core which is responsible for changing the contents of the memory buffer to present the double of their values considering they are equal to the value A7h. The initial and final addresses of a memory buffer are 19h and 62h, respectively.

Solution



Address area	Mnemonic	Arguments	Comments
	MOV	R0, #19h	; Initialize the content of the R0 register with the ; initial address of the memory buffer
	MOV	R2, #4Ah	; Qty. of elements of the memory buffer to be ; analyzed ($62h - 19h + 1 = 4Ah = 74_{10}$). These two ; previous instructions are responsible for ; defining the memory buffer
ADR2	MOV	A, @R0	; Copy the content of the memory location which ; is given by the content of the R0 register (the ; data contained in the memory buffer) to the ; accumulator (A) register
	CLR	C	; Reset the carry-bit flag (C) so that it does not ; influence the result of the subtraction ; instruction
	SUBB	A, #0A7h	; Subtract the content of the accumulator (A) ; register of the carry-bit flag (C) and value 0A7h ; (we must write 0 in front of all hexadecimal ; numbers that begin with letters: A-F) in order to ; identify if the numbers are different from the ; value A7h (#0AFh). If the numbers are different ; from the value A7h, the contents of the memory ; buffer must be unchanged
	JNZ	ADR1	; if the $(Z)=0$ [$(A)\neq 0$ or $(A)\neq A7h$] ; then we must not calculate ; the double of its value
	MOV	A, @R0	; If the content of the memory buffer is equal to ; A7h, we must calculate the double of its ; values and store it in the content of its own ; memory address. So, we must copy the content ; of the memory location whose address is given ; by the content of the R0 register (the data ; contained in the memory buffer) to the ; accumulator (A) register again. This is done ; because the content of accumulator (A) register ; was changed by the previous subtraction ; instruction
	ADD	A, A	; Add the content of the accumulator (A) register ; with itself (multiply by 2), and store the result in ; the accumulator (A) register
	MOV	@R0, A	; Store the double of its value in the same ; memory address of the memory buffer. Change ; the content of the memory buffer with the ; double of its value. The content of the ; accumulator (A) register is copied to the content ; of the memory location whose address is given ; by the content of the R0 register

(continued)

<i>ADRI:</i>	INC	R0	; This instruction adds one unit to the content ; of the memory location R0 register in order to ; address the next memory location of the ; memory buffer to be analyzed (point to the next ; address of the other data to be analyzed)
	DJNZ	R1, <i>ADR2</i>	; This instruction decreases one unit from the ; content of the R1 register (qty. of elements of ; the memory buffer to be analyzed). If the ; quantity of elements of the memory buffer to be ; analyzed is different from zero, the software ; must jump to ADDR2 in order to repeat the ; same procedure performed to the previous data ; considered
	END		; End

4.7 Proposed Problems

- 4.7.1. Implement the flowchart and source program in Assembly regarding an 8051 core microcontroller which is responsible for subtracting the content of the memory location whose address is given by content of the R1 register and the content of the memory location whose address is 3Ah. Besides, the result must be rotated 3 bits to the left and added to the carry-bit with the content of the R5 register. The result must be stored in the content R1 register of the penultimate register bank.
- 4.7.2. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which determines the quantity of numbers which are smaller than 6Dh and present bit 4 equal to 1 of a memory buffer whose initial and final addresses are 2Eh and 74h, respectively. The quantity of elements with these characteristics must be stored in the content of the memory location whose address is 34h.
- 4.7.3. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which calculates the quantity of numbers which are different from the value AAh, present even parity, and are odd numbers of a memory buffer whose initial and final addresses are 3Ch and 61h, respectively. The quantity of elements with these characteristics must be stored in the content of the R3 register of the second bank of registers (bank 1).
- 4.7.4. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which calculates the quantity of numbers which are different from the value AAh, are higher than the content of the R2 register, and present bit 2 equal to 0 of a memory buffer whose initial and final addresses are 4Fh and 78h, respectively. The quantity of elements with these characteristics

must be stored in the content of the R7 register of the third bank of registers (bank 2).

- 4.7.5. Implement the flowchart and source program in Assembly for the 8051 core microcontroller which is responsible for changing the contents of the memory buffer to present the triple of their values considering they are different from the content of the memory location whose address is equal to 66h and smaller than or equal to 22h. The initial and final addresses of a memory buffer are 19h and 62h, respectively.
-

References

1. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
2. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
3. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
4. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
5. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
6. Atmel Corporation (1997) AT89 Series Hardware Description (0499B-B), Dec 1997
7. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
8. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D-MICRO), June 2008
9. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
10. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

Subroutine and Structuring of the Assembly Programming Language

5.1 Introduction

This chapter structures the Assembly programming language using subroutines discussed in Chap. 1 (it is recommended that the readers revisit Sects. 1.11, 1.12, 1.13, 1.14, and 1.15) [1].

The benefits of the Assembly programming language structuring are manifold and will help you design extremely efficient software in Assembly with a reduced program memory expense and faster. Besides, by learning the concepts of structured programming using the Assembly programming language, the reader will be able to easily learn any other structured language, especially the C language [1].

5.2 Definition of the Stack and Queue to Be Used in the Volatile Memory

Before describing the process of calling and returning of subroutines of an 8051 core microcontroller, two fundamental concepts are presented. The first one is related to the stack, and the second one is related to the queue [1].

Stack is an area of data memory (internal or external RAM) where data can be stored. The principle of storing and reading data from this area of memory follows the concept of physical stacking of things and objects, as humans are used to doing every day. If you reflect about the process of stacking objects or things, notice that the first object or thing stored in a pile (stack) will be the last object or thing to be taken away. If this principle is not followed, the stack is destroyed. This process is called LIFO (last in, first out). If you still have any questions about this, for example, consider a stack consisting of a single column of cooking oil cans on a supermarket shelf. The first can of oil stored in this stack can only be removed last, because if this procedure is not followed, the oil can stack is destroyed. Similarly, the last can of oil stored in that heap (the oil can be located on the top of that stack) should be the first one to be removed so it will not be destroyed. Another feature of a stack regarding

the 8051 core microcontrollers is that it grows in an ascending direction towards the RAM addressing, as a data is stored therein, and decreases in the descending direction towards the RAM addressing, when the data is read from the stack [1–10].

In addition to the concept of stack, another way of storing data in RAM is through queue concept, i.e., in which the first data to be written into the memory must be the first to be read from the memory. This procedure is known as first in, first out (FIFO). To illustrate, one can exemplify with the infamous queues of the banking service on payday. The first person in the queue is the first person to be served. This type of storage is generally used in serial data communication, where the first data stored in the queue is the first to be sent through the serial communication interface [1–10].

5.3 The Special Function Register Entitled Stack Pointer (SP)

The stack pointer (SP) register consists of 8 bits and has the function of storing the RAM address on the top of the stack. The initial address of the stack pointer, after a reset regarding the 8051 core microcontroller, is 07h (last address of the first bank of registers, which corresponds to the address of the R7 register of the first register bank). So, if the value of the stack pointer (SP) is not defined at the beginning of the software changing its default value (07h) for another safe memory location, which is not being used, the programmer cannot use memory locations from the second bank of registers (from the address 08h) because the stack grows in an ascending direction towards the RAM addressing. Therefore, the programmer must properly specify the initial value of the stack pointer to prevent that the stack data overwrite the values of the system variables to be controlled, resulting in malfunctions in the computer system. Usually, at the beginning of the main program, this value should be redefined by the programmer with a RAM address pointing to an address just after the variables area used by the computer system [1–10].

The instructions pertaining to the family of 8051 core microcontrollers that use the stack to store and retrieve data of RAM are *PUSH direct* (stores a byte in the stack), *POP direct* (reads a byte from the stack), *ACALL address11* (performs a subroutine located at the memory address address11), *LCALL address16* (performs a subroutine located at the memory address address16), *RET* (returns from a subroutine), and *RETI* (returns from a service subroutine to an interrupt source) [1].

5.4 Process of Access (Writing and Reading) a Byte Regarding the Stack

The *PUSH direct* and *POP direct* instructions are responsible for storing data in the stack and reading data from the stack, respectively. The argument of these statements is *direct*, which corresponds to the address of the content of a register (the address 00h must be indicated in these instructions if the content of the R0 register of the first register bank is used) or memory location (direct addressing) [1–10].

Table 5.1 Description of *PUSH direct* and *POP direct* instructions

Instruction	Bytes	Cycles	Code	Operation
PUSH direct	2	2	1100 0000 direct address	$(PC) \leftarrow (PC) + 2$ $(SP) \leftarrow (SP) + 1$ $((SP)) \leftarrow (direct)$
POP direct	2	2	1101 0000 direct address	$(PC) \leftarrow (PC) + 2$ $(direct) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$

Table 5.1 describes the features of the *PUSH direct* and *POP direct* instructions of the 8051 core microcontroller [1–10].

The *PUSH direct* instruction stores the contents of a register or memory location whose address is defined by direct addressing (value given by *direct*) on the stack. This storage is done by the microprocessor of the microcontroller by the following microinstructions [1–10]:

- First, the content of the program counter (PC) register is added to two units in the content of the PC register $[(PC) \leftarrow (PC) + 2]$, i.e., it points to the address of the next instruction to be fetched and executed by the microprocessor of the microcontroller, since the direct *PUSH instruction* occupies two program memory addresses.
- Then, the microprocessor adds one unit to the content of the stack pointer $[(SP) \leftarrow (SP) + 1]$. So, for example, if the content of the stack pointer (SP) is equal to 07h (default value), it will point to the address 08h of internal RAM (first address of the second register bank, which corresponds to the address of the R0 register of the second register bank).
- Next, the content of the memory location whose address is *direct* is stored in the content of the memory location of the internal RAM (where the stack was defined). Its address is given by the content of the stack pointer register $[((SP)) \leftarrow (direct)]$.

In short, the *PUSH direct* instruction calculates the address of the next instruction to be fetched and executed by the microprocessor of the microcontroller. It points to the next RAM address where the stack was defined, and it finally stores a byte in the stack by using a mixed or mixed addressing (indirectly by registered, $((SP))$ and direct, $(direct)$) [1–10].

The direct *POP direct* instruction reads a byte of the stack. This is done by the microprocessor of the microcontroller through the following microinstructions [1–10]:

- First, the content of the program counter (PC) register is added by two units $[(PC) \leftarrow (PC) + 2]$, i.e., it points to the address of the next instruction to be fetched and executed by the microprocessor of the microcontroller, since this instruction occupies two program memory addresses.
- Then, the content of RAM which the stack was defined, whose address is given by the content of the stack pointer (SP), is copied to the content of the register or memory location whose address is given by direct $[(direct)]$.

Stack Pointer (SP)	Addresses of the internal RAM where the stack has been created	Contents of the internal RAM
50 _h →	50 _h	10 _h
	51 _h	11 _h
	52 _h	12 _h

Fig. 5.1 Addressing range of the internal RAM which the stack has been created

- Afterwards, the microprocessor of the microcontroller decrements the content of the stack pointer by one unit $[(SP) \leftarrow (SP)-1]$. Therefore, if the content of the stack pointer content is equal to 08_h, it is changed to the value 07_h.

In brief, *POP direct* calculates the address of the next instruction to be executed; reads a byte of the stack, using mixed or direct (direct) and indirect ((SP)) addressing modes; and finally decrements the stack pointer (SP) by one unit [1–10].

It is important to highlight that after executing a *PUSH direct* and a *POP direct* instruction, the content of the stack pointer (SP) returns to its initial position [1–10].

To illustrate, consider initially that the content of the stack pointer (SP) has been initialized with the value 50_h (definition of the initial address of the stack), the content of the R1 register of the first register bank is equal to 23_h, and contents of the memory locations of the internal RAM, whose addresses are 50_h, 51_h, and 52_h, are equal to 10_h, 11_h, and 12_h, respectively, according to Fig. 5.1 [1–10].

If we want to store the content of the R1 register of the first register bank (register bank 0) in the stack, the *PUSH 01h* instruction must be executed (01_h is the address of the memory location of the R1 register of the first register bank, i.e., register bank 0, because this instruction uses the direct addressing mode). Then, we have:

1. $(PC) \leftarrow (PC) + 2$: As (PC) has not been defined, this microinstruction is responsible for pointing to the program memory address of the next instruction to be fetched and executed by the microprocessor of the 8051 microcontroller.
2. $(SP) \leftarrow (SP) + 1 = 50h + 1 = 51h$: This microinstruction is responsible for pointing to the subsequent address of the internal RAM, where the stack has been defined, whose content of the stack pointer was initially set to 50_h.
3. $((SP)) = (51h) \leftarrow (01h) = (R1) = 23h$: In the content of the internal RAM where the stack has been created, whose address given by the content of the stack pointer (SP) register $[(SP) = (51h)]$ is copied with the content of the R1 register (23_h), as illustrated in Fig. 5.2.

Some conclusions can be drawn after executing the *PUSH 01h* instruction, as follows:

1. The content of the memory location of the internal RAM, where the stack has been defined, whose address is given by the initial value of the stack pointer (SP) register, remains with its original value, i.e., it has not been changed by this instruction $[(50h) = 10h]$.

Stack Pointer (SP)	Addresses of the internal RAM where the stack has been created	Contents of the internal RAM
	50 _h	10h
51 _h →	51 _h	23 _h
	52 _h	12h

Fig. 5.2 Final conditions of the content of the SP register and of the content of the stack after the PUSH 01h instruction

Stack Pointer (SP)	Addresses of the internal RAM where the stack has been created	Contents of the internal RAM
50 _h →	50 _h	10h
	51 _h	23h
	52 _h	12h

Fig. 5.3 Final conditions of the content of the SP register and of the content of the stack after the POP 01h instruction

2. The content of the stack point (SP) register points to the RAM address at the top of the stack, and its contents are now 51h.
3. The content of R1 register (23h) has been stored in the internal RAM whose address is given by the content of the stack pointer (SP) register [$((SP)) = (51h)$].

Therefore, the stack has grown by one unit in the ascending direction towards the internal RAM addressing.

Now, by executing the POP 01h instruction, we have:

1. $(PC) \leftarrow (PC) + 2$: As (PC) has not been defined, this microinstruction is responsible for pointing to the program memory address of the next instruction to be fetched and executed by the microprocessor of the 8051 microcontroller.
2. $(01h) = (R1) \leftarrow ((SP)) = (51h) = 23h$: In the content of R1 register is copied with the content of the internal RAM (stack), whose address is given by the content of the stack point (SP) register.
3. $(SP) \leftarrow (SP) - 1 = 51h - 1 = 50h$: This microinstruction subtract one unit of the contents of the stack pointer (SP) register, i.e., the content of the stack pointer (SP) register, which defines the initial address of the stack, returns to its initial value (50h).

Figure 5.3 illustrates the final conditions of the content of the stack pointer (SP) register and the contents of the internal RAM where the stack has been defined, after executing the POP 01h instruction.

It is important to highlight that after the execution of the PUSH 01h and POP 01h instructions, the value of the content of the stack pointer (SP) returns to its initial condition, which in this case is equal to 50h. Besides, the data that was stored in the stack continues stored in the content of the address 51h of the internal RAM, where the stack has been defined, after executing the POP 01h instruction. It stays there until the stack is used again. The stack decreases in a descending direction towards

the internal RAM addressing after executing the POP 01h instruction. Additionally, the content of the memory location whose address is given by the content of the initial value of the stack pointer (SP) register, which defines the stack in the internal RAM, is never used by the instructions PUSH direct and POP direct. Furthermore, we can observe that the stack decreases in a descending direction towards the internal RAM addressing.

If a subroutine needs to use some registers or memory locations that the main program is using, we must usually use some *PUSH direct* instructions in the beginning of a subroutine to save the values of the registers or memory positions in the stack. In the end of the subroutine, before the RET instruction (return of the subroutine), we must retrieve the values of these registers or the memory positions of the stack through the execution of some *POP direct* instructions in order to avoid changing the data of these registers or memory positons, which the main program was considering as the subroutine needs to use them. For instance, if you use the instructions PUSH 00h, PUSH 01h, and PUSH 02h (save the contents of the R0, R1, and R2 registers of the first register bank in the stack), you must use in the end of the subroutine, before the instruction RET, the instructions POP 02h, POP 01h, and POP 00h, respectively, because the stack is based on the LIFO (last in, first out) approach, in order to retrieve the content of the R2, R1, and R0 registers.

5.5 The Process of Calling and Returning of a Subroutine

As discussed in Chap. 1, from Sects. 1.11, 1.12, 1.13, 1.14, and 1.15, the subroutines must be allocated in a region of the program memory separated from the main program memory. Thus, based on the flowchart of Fig. 1.35, presented in Chap. 1, whenever the use of subroutines is discussed, it is possible to associate it with a program structure that has three parts. The first program memory area must contain the different subroutines to attend the objective of the computer system, the second region of the memory program must contain the hardware programming and initialization of the system variables to be controlled, and the third area of the memory program (main program) must contain the calls of different subroutines [1–10].

Regarding the memory region of the main program (third part of the memory program/software), and considering the 8051 core microcontrollers, the instructions for the calling of the different subroutines that can be used are the instruction *ACALL addr11* [the initial address of the subroutine can be in a maximum distance of 1024 ($=2^{11}$) bytes above or below this instruction] and the instruction *LCALL addr16* [the initial address of the subroutine can be at a distance of 65,536 ($=2^{16}$) bytes above or below this instruction]. These two instructions of calling the subroutines must be located in the main program, as their functions are to call and execute the subroutines that are in the first region of the program memory. They are responsible for transferring the processing flow from the main program region to the memory area which the different subroutines are written [1–10].

Every subroutine is characterized by its initial address (*addr16*) and by the instruction RET (subroutine return), which must be the last instruction of the subroutine. Its function is to return the processing flow to the main program [1–10].

Table 5.2 Description of the instructions *ACALL addr₁₁* and *LCALL addr₁₆*

Instruction	Bytes	Cycles	Code	Operation
ACALL addr ₁₁	2	2	a ₁₀ a ₉ a ₈ 10001 a ₇a ₀	(PC) \leftarrow (PC)+2 (SP) \leftarrow (SP) + 1 ((SP)) \leftarrow (PC ₇₋₀) (SP) \leftarrow (SP) + 1 ((SP)) \leftarrow (PC ₁₅₋₈) (PC ₁₀₋₀) \leftarrow addr ₁₁
LCALL addr ₁₆	3	2	0001 0010 a ₁₅a ₈ a ₇a ₀	(PC) \leftarrow (PC)+3 (SP) \leftarrow (SP) + 1 ((SP)) \leftarrow (PC ₇₋₀) (SP) \leftarrow (SP) + 1 ((SP)) \leftarrow (PC ₁₅₋₈) (PC ₁₅₋₀) \leftarrow addr ₁₆

The detailed description of these two instructions of calling the subroutines is described in Table 5.2 [1–10].

Initially, the *ACALL addr₁₁* instruction (Absolute call) increments the contents of the program counter (PC) register with two units, as it occupies two contents of the program memory, i.e., the microprocessor calculates the address of the subsequent instruction to the instruction *ACALL addr₁₁*. This address corresponds to the return address of the subroutine, i.e., the address of the subsequent instruction to the subroutine call (*ACALL addr₁₁*), which must be executed after executing the subroutine. Therefore, the content of the program counter (PC) register includes the return address of this subroutine. This return address is stored in the stack so that the instruction RET knows where it must return to the main program after running the subroutine. In this way, the instruction *ACALL addr₁₁* stores the contents of the PC (16-bit) register in the stack by using indirect addressing mode by register. This is done in two parts. First, the content of the SP register is incremented by one unit; then the least significant byte of the program counter (PC) register (PCL) is stored in the stack, whose address is given by the contents of the stack pointer. Afterwards, the content of SP register is added to one unit again in order to address the next address where the most significant byte of the content of the PC register will be stored. Finally, the most significant byte of the content of the program counter (PC) register (PCH) is stored in the stack in the memory location whose address is given by the content of the stack pointer (SP) register. It is important to highlight that the subroutine must be located at a maximum distance of 1 Kbyte above or below the *ACALL addr₁₁* instruction. No flag of the content of the program status word (PSW) register is affected by this instruction [1–10].

To illustrate how the instruction *ACALL addr₁₁* works, consider that the content of the stack pointer (SP) register is equal to 07h, the *ACALL subrtn* statement is stored in the content of the program memory location, whose address is equal to 0123h, *subrtn* defines the initial address of the subroutine, that in this case is equal to 0345h, and the contents of the internal RAM where the stack was defined are equal to XXh, YYh, and ZZh, respectively, as indicated in Fig. 5.4.

Program memory address	Mnemonic	Argument
0123h	ACALL	subrtn
	or	
0123h	ACALL	0345h

Stack Pointer (SP)	Address of internal RAM	Content of the internal RAM
07h →	07h	XXh
	08h	YYh
	09h	ZZh

Fig. 5.4 Initial conditions of the (PC), (SP) and contents of the internal RAM where the stack was defined by (SP)

Thus, executing the instruction *ACALL addr11* (*ACALL 0345h*, because *addr11* represents the address 0345h) instruction, we have:

Program memory address	Mnemonic	Argument	Microinstructions
0123h	ACALL	0345h	<p>Calculate the return address of the subroutine (address of the subsequent instruction to the instruction ACALL 0345h)</p> $(PC) \leftarrow (PC) + 2 = 0123h + 2 = 0125h$ $[(PCH) = 01h \text{ and } (PCL) = 25h]$ <p>Store the return address of the subroutine in the stack:</p> $(SP) \leftarrow (SP) + 1 = 07h + 1 = 08h$ [add one unit to the content of the stack pointer (SP) register] <p>$((SP)) = (08h) \leftarrow (PC_{7-0}) = (PCL) = 25h$ (first, it stores the least significant byte of the content of the program counter (PC) register in the stack, in the address of the internal RAM given by the content of the stack pointer (SP) register)</p> $(SP) \leftarrow (SP) + 1 = 08h + 1 = 09h$ [add one unit to the content of the stack pointer (SP) register] <p>$((SP)) = (09h) \leftarrow (PC_{15-8}) = (PCL) = 01h$ (after, it stores the most significant byte of the content of the program counter (PC) register in the stack, in the address of the internal RAM given by the content of the stack pointer (SP) register)</p> <p>Jump to the initial address of the subroutine</p> $(PC_{10-0}) = (PC) \leftarrow addr11 = 0345h$ (it changes the processing flow of the main program to run the subroutine from the initial address of the subroutine (0345h))

Stack Pointer (SP)	Address of internal RAM	Content of the internal RAM where the stack was defined
	07h	XXh
	08h	25h [=PCL]]
09h →	09h	01h [=PCH]]

Fig. 5.5 Final conditions of the content of the SP register and of the contents of the internal RAM where the stack was defined after the execution of the instruction ACALL 0345h

Figure 5.5 illustrates the final conditions of the contents of the stack pointer (SP) register and addresses 08h and 09h, respectively (contents of the internal RAM memory where the stack was defined), after the execution of the instruction *ACALL 0345h*.

Observe that after performing the instruction *ACALL subrotn* (*ACALL 0345h*), we have:

- The content of the stack pointer (SP) register is added to two units (09h) because the return address is composed of two bytes [01h and 25h] and the stack grows upwards from the addressing of the internal RAM.
- The contents of the memory locations 08h and 09h have the values 25h (least significant byte of the return address of the subroutine) and 01h (most significant byte of the return address of the subroutine), respectively.
- The content of the program counter register (PC) becomes equal to 0345h, which corresponds to the subroutine's starting address. Then, the processing flow is transferred from the main program to the subroutine.

The other instruction that is capable of jumping unconditionally to the subroutine's start address is the *LCALL addr16* (Long call), as *addr16* corresponds to a 16-bit address. This instruction is able to jump to the subroutine regarding a maximum distance of 64 Kbytes from it [1–10].

Initially, the instruction *LCALL addr16* increments the content of the program counter (PC) register by three units. This occurs because this instruction is defined by 3 bytes (it occupies three contents of the program memory and consequently uses three addresses of the memory program). This is done to calculate the subsequent address to the instruction *LCALL addr16*. This address corresponds to the return address of the subroutine. Therefore, the content of the program counter (PC) register contain the return address of the subroutine. Afterwards, the instruction *LCALL addr16* stores the content of the program counter (PC) register in the stack by using indirect addressing mode, taking into account the content of the stack pointer (SP) register. This information will be used by the RET instruction of the subroutine so that it can go back to the main program. The process to do this is as follows: first, the content of the stack point (SP) register is incremented with one unit; then, the least significant byte of the content of the program counter (PC) register (PCL) is stored in the stack by using indirect addressing mode regarding the content of the

stack pointer (SP) register; subsequently, the content of the stack pointer (SP) register is added to one unit again; thereafter, the most significant byte of the content of the program counter (PC) register (PCH) is stored in the stack in the memory location whose address is given by the content of the stack pointer (SP). Finally, the content of the program counter (PC) register is initialized with the initial address of the subroutine ($addr_{16}$). This causes the execution of the subroutine, i.e., the processing flow is changed from the main program to the subroutine. No flag of the content of the program status word (PSW) register is affected by this instruction [1–10].

To illustrate the operation of the instruction $LCALL\ addr_{16}$, consider that the content of the stack pointer (SP) register is 07h, the instruction $LCALL\ subrtn$ is stored in the content of the program memory location whose address is 0123h, $subrtn$ corresponds to the memory address 1234h, and the contents of the internal RAM where the stack was defined are equal to XXh, YYh, and ZZh, respectively, according to Fig. 5.6.

By executing the instruction $LCALL\ addr_{16}$ ($LCALL\ 1234h$), we have:

Program memory address	Mnemonic	Argument	Microinstructions
0123h	LCALL	1234h	<p>Calculate the return address of the subroutine (address of the subsequent instruction to the instruction $ACALL\ 1234h$) $(PC) \leftarrow (PC) + 3 = 0123h + 3 = 0126h$ $[(PCH) = 01h \text{ and } (PCL) = 26h]$</p> <p>Store the return address of the subroutine in the stack:</p> <p>$(SP) \leftarrow (SP) + 1 = 07h + 1 = 08h$ [add one unit to the content of the stack pointer (SP) register]</p> <p>$((SP)) = (08h) \leftarrow (PC_{7..0}) = (PCL) = 26h$ (first, it stores the least significant byte of the content of the program counter (PC) register in the stack, in the address of the internal RAM given by the content of the stack pointer (SP) register)</p> <p>$(SP) \leftarrow (SP) + 1 = 08h + 1 = 09h$ [add one unit to the content of the stack pointer (SP) register]</p> <p>$((SP)) = (09h) \leftarrow (PC_{15..8}) = (PCH) = 01h$ (then, it stores the most significant byte of the content of the program counter (PC) register in the stack, in the address of the internal RAM given by the content of the stack pointer (SP) register)</p> <p>Jump to the initial address of the subroutine</p> <p>$(PC_{15..0}) = (PC) \leftarrow addr_{16} = 1234h$ (changes the processing flow of the main program to run the subroutine from the initial address of the subroutine (1234h))</p>

Program memory address	Mnemonic	Argument
0123h	LCALL	subrtn
	ou	
0123h	LCALL	1234h

Stack Pointer (SP)	Address of internal RAM	Content of the internal RAM
07h →	07h	XXh
	08h	YYh
	09h	ZZh

Fig. 5.6 Initial conditions of the (PC), (SP) and contents of the internal RAM where the stack was defined by (SP)

Stack Pointer (SP)	Address of internal RAM	Content of the internal RAM
	07h	XX _h (any data)
	08h	26h
09h →	09h	01h

Fig. 5.7 Final conditions of the content of the SP register and of the contents of the internal RAM where the stack was defined after the execution of the instruction *LCALL 1234h*

Figure 5.7 illustrates the final conditions of the contents of the stack pointer (SP) register and addresses 08h and 09h, respectively (contents of the internal RAM memory where the stack was defined), after the execution of the instruction *LCALL 1234h*.

Observe that after executing the instruction *LCALL 1234h*, we have:

- The content of the stack pointer (SP) register is added to two units (09h) because the return address is composed of two bytes [01h and 26h] and the stack grows upwards from the addressing of the internal RAM.
- The contents of the memory locations 08h and 09h have the values 26h (least significant byte of the return address of the subroutine) and 01h (most significant byte of the return address of the subroutine), respectively.
- The content of the program counter (PC) register changes to the value 1234h, which corresponds to the subroutine's starting address, and the processing flow is transferred from the main program to the subroutine.

The process by which the microprocessor of the 8051 core microcontroller returns from a subroutine to the main program is based on the recovery of the return

address (two read operations), which was stored by the instruction *ACALL addr11* or *LCALL addr16*. This is done through the RET (subroutine return) instruction. Thus, the RET is responsible for reading two successive bytes of the stack that compose the return address of the subroutine, using indirect addressing and taking into account the content of the stack pointer (SP) register, storing them in the contents of the program counter (PC) register.

First, the most significant byte of the return address is read from the stack and stored in the content of the PCH register; then the least significant byte of the return address is read from the stack and stored in the content of the PCL register, respecting the principle of stack storage (LIFO). This instruction is executed by the microprocessor through the read operation of the content of the memory location whose address is given by the content of the stack pointer (SP) register and by a writing operation in the most significant byte of the content of the program counter (PCH) register. Then, the content of the stack pointer (SP) register is decremented by one unit, and again the microprocessor performs a read operation of the content of the memory location whose address is given by the content of the stack pointer (SP) register. It stores this value in the least significant byte of the program counter (PCL) register. Thereafter, the content of the stack pointer (SP) register is again decremented by one unit. Thus, as the content of the PC register contains the return address of the subroutine, the next instruction to be executed is the one that comes immediately after the instruction *ACALL addr11* or *LCALL addr16*. Therefore, the return of the subroutine is performed, i.e., the processing flow is transferred from the subroutine to the main program, by executing the subsequent instruction to the *ACALL addr11* or *LCALL addr16*. No flag of the content of the program status word (PSW) register is affected by this instruction. Table 5.3 describes the functionality of the RET instruction [1–10].

To describe how the RET instruction works, initially consider that the content of the stack point (SP) register is equal to 09h and the contents of the memory locations

Table 5.3 Description of the RET instructions

Instruction	Bytes	Cycles	Code	Operation
RET	1	2	0010 0010 ₂	$(PC_{15-8}) \leftarrow ((SP))$ $(SP) \leftarrow (SP)-1$ $(PC_{7-0}) \leftarrow ((SP))$ $(SP) \leftarrow (SP)-1$

of the internal RAM where the stack is defined, whose addresses 08h and 09h are equal to 23h and 01h, respectively, according to Fig. 5.8 [1–10].

If the RET instruction is executed, considering that it is in the program memory address 26AFh, we have:

Program memory address	Mnemonic	Argument	Microinstructions
26AFh	RET		<p>$(PC_{15-8}) = (PCH) \leftarrow ((SP)) = (09h) = 01h$: it reads the most significant byte of the return address of the subroutine that was stored by the instruction <i>ACALL addr11</i> or <i>LCALL addr16</i> and stores it in the most significant byte of the program counter (PC) register (PCH).</p> <p>$(SP) \leftarrow (SP)-1 = 09h-1 = 08h$: it decrements the content of the stack pointer (SP) register by one unit</p> <p>$(PC_{7-0}) = (PCL) \leftarrow ((SP)) = (08h) = 23h$: it reads the least significant byte of the return address of the subroutine that was stored by the instruction <i>ACALL addr11</i> or <i>LCALL addr16</i> and stores it in the least significant byte of the program counter (PC) register (PCL). Therefore, the program flow goes back to run the main program the subsequent instruction after the instruction <i>ACALL addr11</i> or <i>LCALL addr16</i></p> <p>$(SP) \leftarrow (SP)-1 = 08h-1 = 07h$: it decrements the content of the stack pointer (SP) register by one unit</p>

Figure 5.9 illustrates the final conditions of the contents of the stack pointer (SP) register and addresses 08h and 09h, respectively (contents of the internal RAM memory where the stack was defined), after the execution of the instruction *LCALL 1234h*.

Stack Pointer (SP)	Address of internal RAM	Content of the internal RAM
	07 _h	XX _h
	08 _h	23 _h
09 _h →	09 _h	01 _h

Fig. 5.8 Initial conditions of the (SP) and contents of the internal RAM where the stack was defined by (SP)

Stack Pointer (SP)	Address of internal RAM	Content of the internal RAM
07h →	07 _h	XX _h
	08 _h	23 _h
	09 _h	01 _h

Fig. 5.9 Final conditions of the content of the SP register and of the contents of the internal RAM where the stack was defined after the execution of the RET instruction

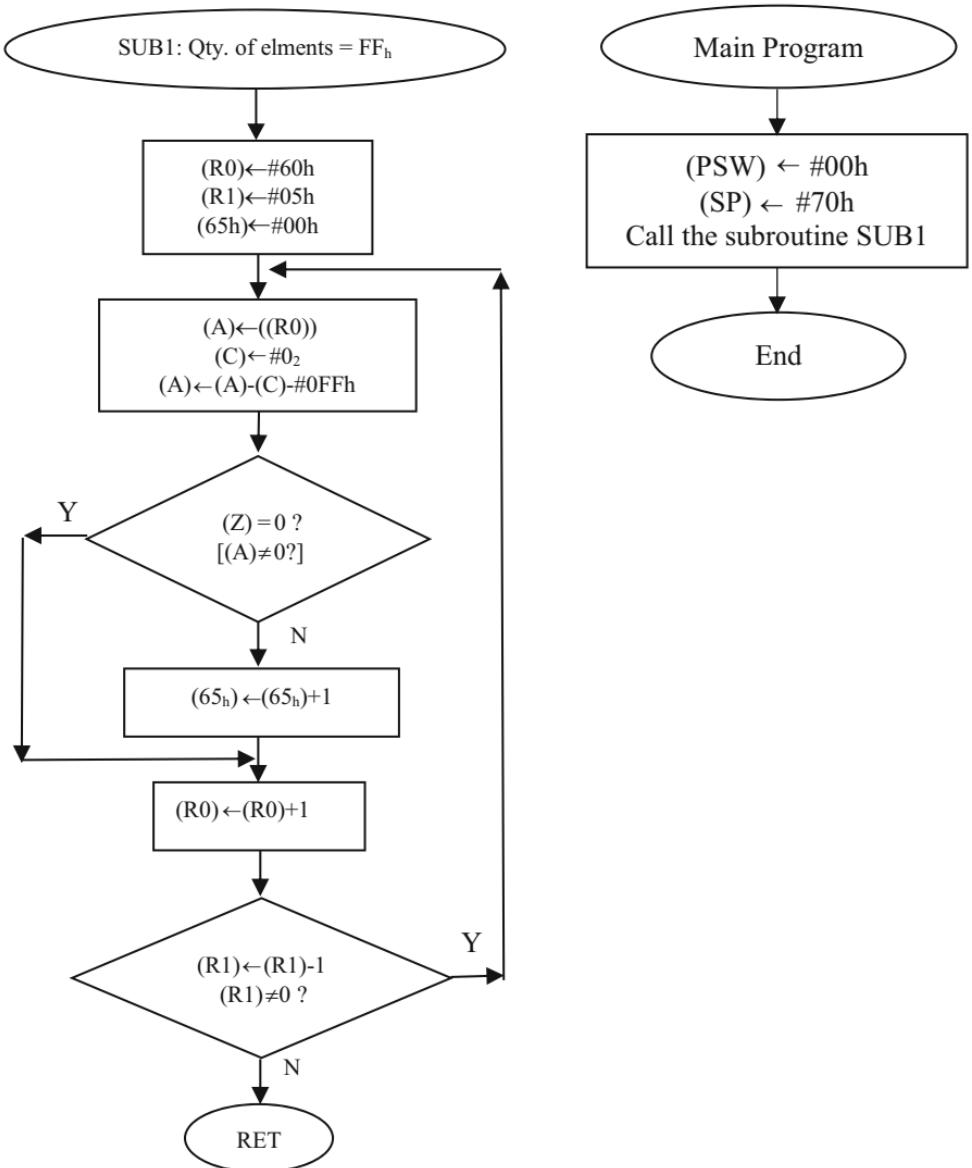
After executing the RET instruction, we can observe that:

- The content of the stack pointer (SP) register will become equal to 07h, i.e., return to the original position which it was before executing the instruction *ACALL addr11* or *LCALL addr16*.
- The content of the program counter (PC) register will become equal to 0123h, which corresponds to the address of the next instruction to be executed in the main program. It is located immediately after the instruction *ACALL addr11* or *LCALL addr16*.
- The contents of the memory locations 08h and 09h remain stored in these memory locations until a new storage in the stack occurs through the instructions *ACALL addr11*, *LCALL addr11*, and *PUSH direct*.

5.6 Resolved Exercises

- 5.6.1. Design a structured program (flowchart and source program using subroutines) in Assembly using the 8051 core microcontroller that calculates the quantity of elements equal to the value FFh of a memory buffer. The initial and final addresses of this memory buffer are equal to 60h and 64h, respectively. The result must be stored in the content of the memory location whose address is 65h.

Solution



Address Region	Mnemonic	Argument	Comments
	<i>ORG</i>	<i>0100h</i>	; programming language guideline in ; Assembly to set the initial address ; (0100h) of the subroutine in the program ; memory
SUB1	MOV	R0, #60h	; Initial address of the memory buffer
	MOV	R1,#05h	; Qty. of elements of the memory buffer
	MOV	65h, #00h	; Counter of the quantity of elements that ; are equal to FFh
ADDR2	MOV	A, @R0	; Copy the data of the memory buffer to the ; accumulator (A) register
	CLR	C	; resets the content of the carry-bit flag so ; as not to influence the result of the ; subtraction operation
	SUBB	A, #0FFh	; Subtracts the (A) from the value FFh (the ; number 0 must be written in front of the ; value FFh because all values that begin ; with the letters from A to F must be ; written with 0 in front of them, otherwise the ; Assembly compiler will indicate ; a compilation error
	JNZ	ADDR1	; If $(A) \neq 0$ [$(A) \neq 0FF_h \Rightarrow (PC) \leftarrow ADDR1$] [it ; jumps to ADDR1 so that it does not add ; one unit to the (65 _h), which contain the ; qty. of elements that are equal to FFh]
	INC	65h	; As $(A) = 0$ [$(A) = 0FF_h$], we must add ; one unit to the (65 _h)
ADDR1	INC	R0	; Points to the next address of the memory ; buffer for its content to be analyzed
	DJNZ	R1, ADDR2	; Decrement by one unit the qty. of ; elements of the memory buffer to be ; analyzed. If it is different from zero (it ; has more data to be analyzed in the ; memory buffer), it jumps to the program ; memory ADDR2 address [$(PC) \leftarrow ADDR2$] ; to carry out the same analyses, which were previously ; performed to the first element of the buffer
	RET		; After analyzing all the elements of the ; memory buffer [when $(R1) = 0$], it returns from this ; subroutine to the main program
	<i>ORG</i>	<i>0200h</i>	; programming language guideline in ; Assembly to set the initial address ; (0200h) of the main program
PROGP	MOV	PSW, #00h	; It resets the flags and register bank (bank 0)
	MOV	SP, #70h	; It defines the initial address of the stack
	ACALL	SUB1	; It calls the subroutine entitled SUB1.
	END		; End of the main program

The guidelines of the Assembly language “ORG address” is used to allocate a subroutine or part of a program in a specific location (defined by “address”) of the memory program. Another way to do this is by using the guidelines:

Code at address

:

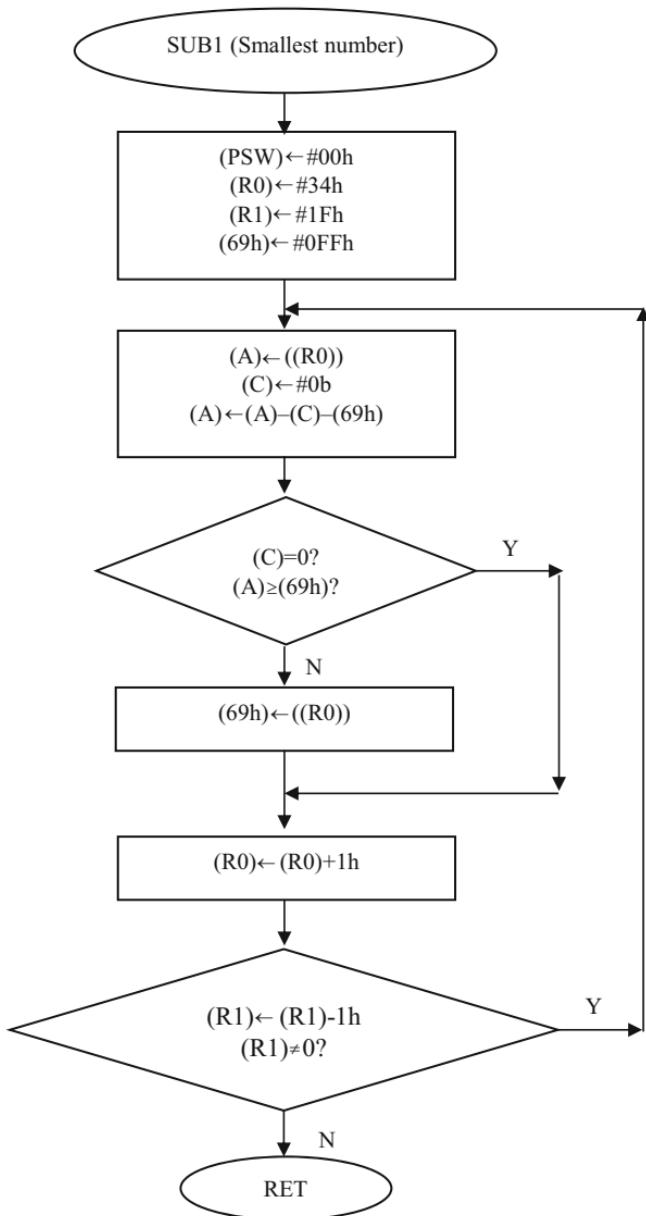
Write the instructions of the subroutine or part of a program here between the Code at address and Code guidelines to define the range of program memory

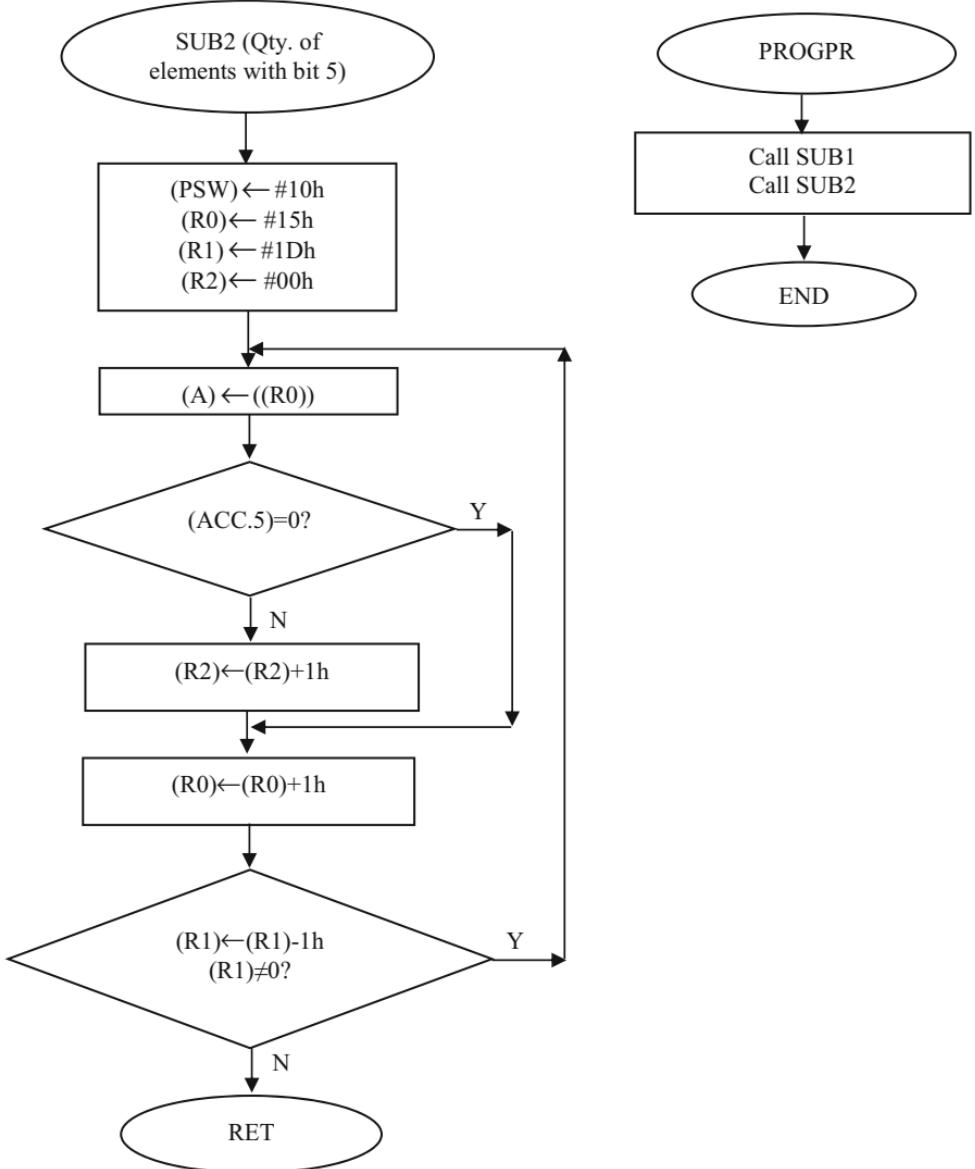
:

Code

- 5.6.2. Design a structured program (flowchart and source program) in Assembly using the 8051 core microcontroller that must determine the smallest number of memory buffer. The initial and final addresses of this memory buffer are equal to 34h and 52h, respectively. The result must be stored in the content of the memory location whose address is equal to 69h. This same software must calculate the quantity of elements which present the bit 5 equal to 1 of another memory buffer. The initial and final addresses of this other memory buffer are equal to 15h and 31h, respectively. This quantity must be stored in the content of the R2 register of the penultimate bank of registers.

Solution





Address region	Mnemonic	Argument	Comments
	<i>ORG</i>	<i>0100h</i>	; programming language guideline in ; Assembly to set the initial address ; (<i>0100h</i>) of the subroutine in the program ; memory
SUB1:	MOV	PSW, #00h	; Define register bank 0

(continued)

	MOV	R0, #34h	; Initial address of the memory buffer
	MOV	R1,#1Fh	; Qty. of elements of the memory buffer
	MOV	69h, #0FFh	; Memory location responsible for storing the smallest ; value of the memory buffer. ; Initially, it is equal to FFh
ADDR2	MOV	A, @R0	; Copy the data of the memory buffer to the ; accumulator (A) register
	CLR	C	; resets the content of the carry-bit flag so ; as not to influence the result of the ; subtraction operation
	SUBB	A, 69h	; Subtracts the (A) from the value FFh (the ; number 0 must be written in front of the ; value FFh because all values that begin ; with the letters from A to F must be ; written with 0 in front of them, otherwise the Assembly ; compiler will indicate a ; compilation error
	JNC	ADDR1	; If $(A) \geq (69h) \Rightarrow (PC) \leftarrow ADDR1$ [it ; jumps to ADDR1 address because this ; instruction must not store a value higher ; than the one which is stored in the ; content of the memory location whose ; address is 69h (location where the smallest value of the ; memory buffer must be stored)
	MOV	69h, @R0	; As $(A) < (69h)$, this instruction must store ; the value of this element of the memory ; buffer because it is smaller than that ; which is stored in the content of the ; memory location whose address is 69h ;(location where the smallest value of the memory buffer ; must be stored)
ADDR1	INC	R0	; Points to the next address of the memory ; buffer for its content to be analyzed
	DJNZ	R1, ADDR2	; Decrement by one unit the qty. of ; elements of the memory buffer to be ; analyzed and if it is different from zero (it ; has more data to be analyzed in the ; memory buffer), it jumps to the program ; memory ADDR2 address [$(PC) \leftarrow ADDR2$] ; to carry out the same analyses which were performed to ; the first element of the buffer
	RET		; After analyzing all elements of the ; memory buffer [when $(R1)=0$], it returns from this ; subroutine to the main program
	ORG	0150h	; programming language guideline in ; Assembly to set the initial address ;(0150h) of the subroutine in the program ; memory
SUB2	MOV	PSW, #10h	; Define the penultimate register bank (2)

(continued)

	MOV	R0, #15h	; Initial address of the memory buffer
	MOV	R1,#1Dh	; Qty. of elements of the memory buffer
	MOV	R2,#00h	; Counter of elements of the memory buffer ; that present the bit 5 equal to 1
ADDR4	MOV	A, @R0	; Copy the data of the memory buffer to the ; accumulator (A) register
	JNB	ACC.5, ADDR3	; If (bit 5)=0 \Rightarrow (PC) \leftarrow ADDR3: this ; instruction must not add one unit to the ; content of the R2 register of the ; penultimate register bank because the ; element of the memory buffer does not ; present the content of its bit 5 equal to 1
	INC	R2	; As (bit 5)=1, this instruction must add ; one unit to the content of the R2 ; register of the penultimate register bank ; because the element of the memory ; buffer presents the content of its bit 5 ; equal to 1
ADDR3	INC	R0	; Points to the next address of the memory ; buffer for its content to be analyzed
	DJNZ	R1, ADDR4	; Decrement by one unit the qty. of ; elements of the memory buffer to be ; analyzed and if it is different from zero (it ; has more data to be analyzed in the ; memory buffer), it jumps to the program ; memory ADDR4 address [(PC) \leftarrow ADDR4] ; to do the same analyses which were previously ; performed to the first element of the buffer
	RET		
	ORG	0200h	; programming language guideline in ; Assembly to set the initial address ; (0200h) of the main program
PROGP	MOV	SP, #70h	; It defines the initial address of the stack
	LCALL	SUB1	; It calls the subroutine entitled SUB1
	LCALL	SUB2	; It calls the subroutine entitled SUB2
	END		; End of the main program

5.7 Proposed Problems

- 5.7.1. Design a structured program (flowchart and source program using subroutines) in Assembly using the 8051 core microcontroller that calculates the quantity of elements which are different from the content of the location memory whose address is equal to 1Dh. The initial and final addresses of this

memory buffer are equal to 2Fh and 78h, respectively. The result must be stored in the content of the register 20h.

- 5.7.2. Design a structured program (flowchart and source program) in Assembly using the 8051 core microcontroller that must determine the highest number of memory buffer. The initial and final addresses of this memory buffer are equal to 46h and 74h, respectively. The result must be stored in the content of the memory location whose address is equal to 22h. This same software must calculate the quantity of elements which are higher than the value 3Ah and smaller or equal to the value A2h of another memory buffer. The initial and final addresses of this other memory buffer are equal to 2Ah and 63h, respectively. This quantity must be stored in the content of the R4 register of the second bank of registers.
- 5.7.3. Design a structured program (flowchart and source program) in Assembly using the 8051 core microcontroller that must transform (change) the memory buffer on even numbers (to reset the least significant bit: bit 0 = 0). The initial and final addresses of this memory buffer are, respectively, equal to 33h and 51h. This same software must calculate the quantity of elements which are higher or equal to the value D1h and present odd parity with another memory buffer. The initial and final addresses of this other memory buffer are equal to 38h and 75h, respectively. This quantity must be stored in the content of the memory location whose address is equal to 77h.
- 5.7.4. Design a structured program (flowchart and source program) in Assembly using the 8051 core microcontroller that must determinate the quantity of elements which are even numbers with the memory buffer. The initial and final addresses of this memory buffer are equal to 29h and 65h, respectively. This quantity must be stored in the content of the memory location whose address is equal to 03h. This same software must calculate the quantity of elements different from the value ABh, and they are odd numbers with another memory buffer. The initial and final addresses of this other memory buffer are equal to 38h and 75h, respectively. This quantity must be stored in the content of the memory location whose address is equal to 2Dh.
- 5.7.5. Design a structured program (flowchart and source program) in Assembly using the 8051 core microcontroller that must determinate the quantity of elements which are positive numbers of the memory buffer. The initial and final addresses of this memory buffer are equal to 1Bh and 57h, respectively. This quantity must be stored in the content of the memory location whose address is equal to 26h. This same software must calculate the quantity of elements equal to the value 10h, and they present the bit 4 equal to 0 of another memory buffer. The initial and final addresses of this other memory buffer are equal to 29h and 51h, respectively. This quantity must be stored in the content of the memory location whose address is equal to 33h.

References

1. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica
2. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
3. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
4. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
5. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
6. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
7. Atmel Corporation (1997) AT89 Series Hardware Description (0499B-B), Dec 1997
8. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
9. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D-MICRO), June 2008
10. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014

Input/Output Ports of 8051 Core Microcontrollers

6.1 Introduction

In this chapter, we will learn how to manipulate some input and output devices that are usually present in computer systems implemented with 8051 core microcontrollers. It is fundamental that the reader rereads Chap. 1, in which some different input and output devices are presented. Thus, the reader can begin to design their first hardware and software projects with microcontrollers by controlling mechanical switches, temperature sensors, keyboards, LEDs, displays, motors, etc. [1–10].

I would like to remind you that the design with microcontrollers is limited only by the limit of your creativity, which means that there are no limits [1–10].

It is important to highlight that the more you learn about this area, the more you will become more enthusiastic to work with it, forcing your creativity to work even more to obtain excellent results [1–10].

6.2 Input and Output Ports

The MCS-51 family of 8051 core microcontrollers from Intel presents four input/output ports which are called P0, P1, P2, and P3. Each port is composed of 8 bits. Each bit can be uniquely identified. For instance, bit 0 of port 0 is called P0.0, bit 3 of port 1 is called P1.3, bit 5 of P2 is called P2.5, bit 7 of P3 is called P3.7, and so on, according to Fig. 6.1 [1–10].

All bits of the input/output ports are bidirectional, that is, they can be programmed individually either as input, to receive information from external input interfaces, or as output, to enable/disable external output interfaces [1–10].

It is possible to perform read and write operations with the input/output ports by using the MOV instructions, for example. Besides, it is also possible to perform read, write, and test operations with each bit of the input/output port individually by using,

Fig. 6.1 Description of the input/output ports of the 8051 core microcontroller

	Bit							
	7	6	5	4	3	2	1	0
P0=	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
P1=	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
P2=	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
P3=	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0

Table 6.1 Description of the alternative functions of some bits that P1 (P1.0 and P1.1) and P3 can assume

Bits of the ports	Alternative function
P1.0 *	T2 (input pin of the external clock of the Timer/Counter 2)
P1.1 *	Caption/trigger of the recharging of the <i>Timer/Counter 2</i>
P3.0	RXD (input bit of data of the serial communication channel)
P3.1	TXD (output bit of data of the serial communication channel)
P3.2	INT0 (input pin of the external interruption 0)
P3.3	INT1 (input pin of the external interruption 1)
P3.4	T0 (input pin of the external clock of the <i>Timer/Counter 0</i>)
P3.5	T1 (input pin of the external clock of the <i>Timer/Counter 1</i>)
P3.6	WR (write control signal of the external data memory)
P3.7	RD (read control signal of the external data memory)

*P1.0 e P1.1 only in 8052/32

for instance, the MOV C, PX_(X = 0-3), MOV PX_(X = 0-3), JB bit, address, JNB bit, address, SETB, CLR instructions, etc. [1–10].

Each bit of a port consists of at least three basic circuits [1–10]:

- A D-type flip-flop or latch which is responsible for storing a bit
- A driver of the electrical current that is capable of exciting an output interface
- An input buffer to receive data from a input interface

The output drivers of ports 0 and 2 and the input buffer of port 0 are used for accessing the external memories, when they are connected to the 8051 core microcontroller. The function of port 0 is to define the least significant byte of the external memory address. When this occurs, the control signal called ALE of the 8051 core microcontroller is set (ALE = 1). This information (least significant byte) is multiplexed with the data bus of the external memory. The function of port 2 is to define the most significant byte of the external memory address throughout the instruction cycle [1–10].

Some of the output drivers of the input buffer of port 1 (P1) and all bits of port 3 (P3) are multifunctional (they have multiple functions), i.e., in addition to being able to work normally as input and output bits, they also can assume alternative functions, as described in Table 6.1 [1–10].

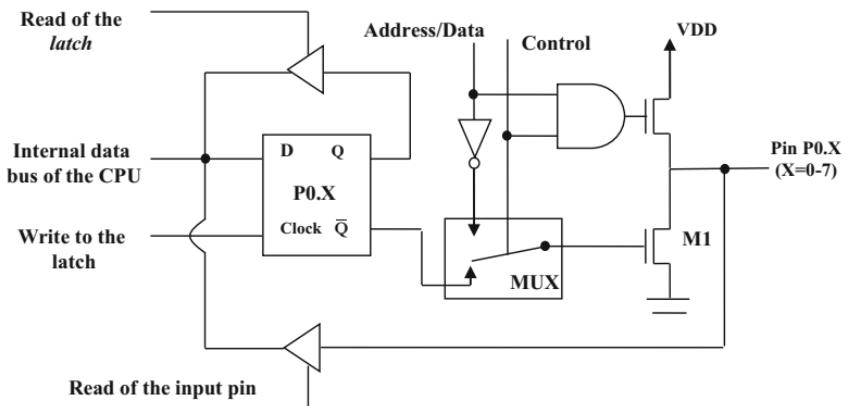


Fig. 6.2 Internal architecture of the bits of P0 of the MCS-51 family of microcontrollers (Intel)

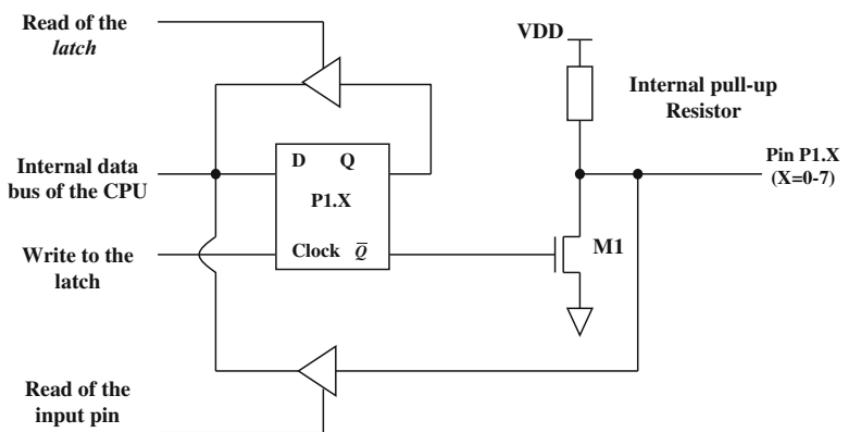


Fig. 6.3 Internal architecture of the bits of the P1 of the MCS-51 family of 8051 core microcontrollers (Intel)

The alternative functions can only be activated if the corresponding latch of the respective port is defined with a logical 1. If the latch is set with logic zero (0), its alternate function is disabled and its function will reproduce the content of the special function register P3 for 8051/31 and P1.0 and P1.1 for 8052/32. In the Timer/Counter mode, the counters of the Timer/Counter 2 are automatically recharged to the RLDH and RLDL registers only if the recharge option is selected ($= 0$) [1–10].

Figures 6.2, 6.3, 6.4, and 6.5 illustrate the functional block diagrams of a typical input and output bit of each one of the four ports in the 8051 core microcontroller family [1–10].

The one-bit latch (one bit of the special function register of a port) is represented as a type-D flip-flop whose clock input is connected to a write signal to the latch coming from the CPU [1–10].

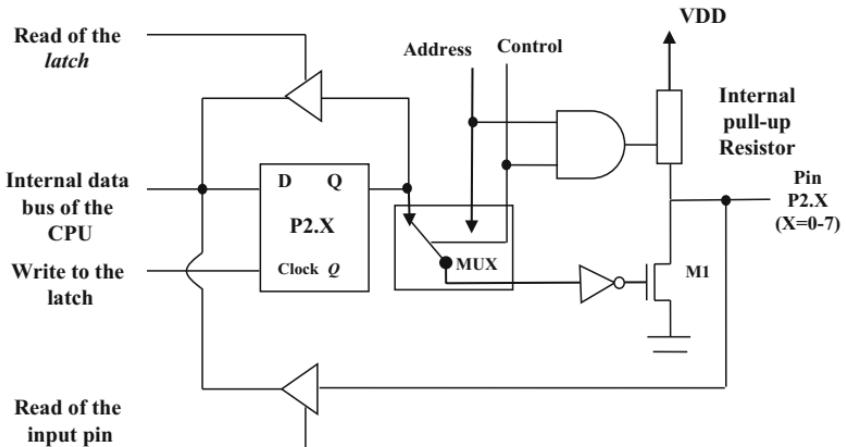


Fig. 6.4 Internal architecture of the bits of the P2 of the MCS-51 family of 8051 core microcontrollers (Intel)

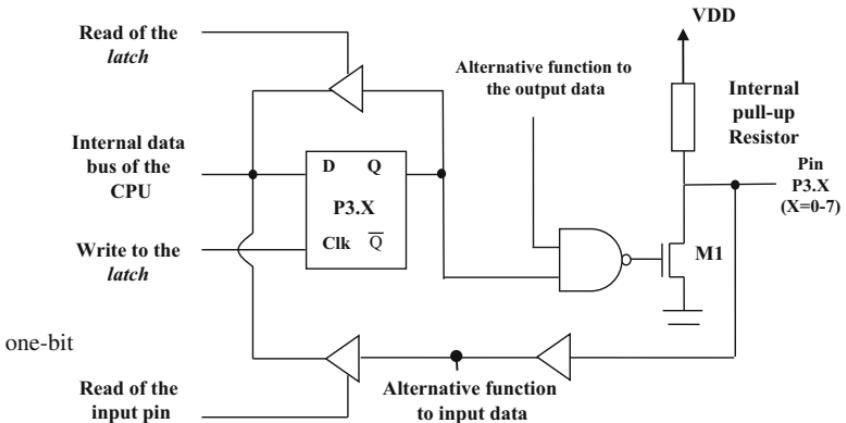


Fig. 6.5 Internal architecture of the bits of the P3 of the MCS-51 family of 8051 core microcontrollers (Intel)

The Q output of the flip-flop is placed on the internal bus in response to a read signal from the latch coming from the CPU [1–10].

The logic level of the port pin is placed on the internal bus in response to a read signal from the pin coming to the CPU [1–10].

Some instructions read latch, others activate the latch read signal, and others activate the pin reading signal [1–10].

Ports 1, 2, and 3 have internal pull-ups. Port 0 has “drain open.” [1–10].

Ports 0 and 2 cannot be used as general purpose ports (input and output) when they are used as address and data bus of the external memories. In order to be used as an input, the respective latch of the ports must present in its output the logical level equal to 1, in which it deactivates the MOSFET of the output current driver [1–10].

Regarding ports 1, 2, and 3, their pins are put in logical levels equal to 1 by an internal pull-up and can also be put in logical level equal to 0 by an external source. The bits of port 0 differ from the others because they do not have internal pull-ups. The MOSFETs connected to the external pull-ups of the P2 output drivers are only used when the port is emitting logic 1 while accessing the external memory, otherwise the MOSFETs that are connected to the pull-ups are turned off, and consequently the lines of P0 which are being used as output lines become open drain type. Writing a logical level equal to 1 on the latches of these bits causes the output MOSFETs to turn off, so the pins float, and thus they can be used as high impedance input bits [1–10].

When a bit of a port considers the information source of a specific instruction that handles the ports, the information is defined by the logical level of the pin itself, rather than the output of the latch. This logic level is always affected by both the microcontroller and the external input. The read value is essentially the OR function of the transistor and the external device. If the external device is in high impedance [gate input of a MOSFET or high output impedance], then reading up an input pin will reflect the previous logical level of the output [1–10].

To use a pin as input, the corresponding output latch should be in a logic level equal to 1. The complemented output (\bar{Q}) of the latch will remain in a logic level equal to 0, and the transistor will behave as an open key. The external device can then supply to the input pin of the input buffer either a high or low logic signal. Therefore, the same port can be used as input and output by writing a logic level equal to 1 on all pins used as input regarding the output operations and bypassing all pins used as output in input operations.

In the instructions of an operand (INC, DEC, DJNZ, and CPL), the output of the latch is used as the data source instead of the input pin. Similarly, the instructions of two operands operating the port as source and as destination (ANL, ORL, and XRL) use the latches' outputs, i.e., they ensure that the latches' bits corresponding to the pins used as inputs will not be done equal to zero in the process of executing these instructions [1–10].

The JBC Boolean instruction tests the output of the latches instead of the input pins. If the latch of P3 presents a logic level equal to 1, then the output level is controlled by the signal called alternative output function. The pin level P3.X (X = 0–7) is always available for the alternate input function [1–10].

As ports 1, 2, and 3 have fixed pull-ups, they are called quasi-bidirectional ports. When configured as input, they go to a logic level equal to 1 and provide current (IIL) when externally placed in logical 0. However, port 0 is considered to be truly bidirectional, because when configured as input, it floats [1–10].

After a reset signal, all latch outputs of the 8051 core microcontroller family are defined with logic level 1. In this way, all ports are programmed as input. If a logical level equal to zero is subsequently written to a latch of a port, it is programmed as output. This same port can be reconfigured again as input if a new write operation of a high logic level is performed in this latch of the port again [1–10].

The instructions that they read, modify, and write on the latches are called read-modify-write instructions, such as ANL (e.g., ANL P1, A), ORL (e.g., ORL P2, A),

XRL (e.g., XRL P3, A), JBC (e.g., JBC P1.1, LABEL), CPL (e.g., CPL P3.0), INC (e.g., INC P2), DEC (e.g., DEC P2), DJNZ (e.g., DJNZ P3, LABEL), MOV PX.Y, C (e.g., MOV P0.1, C), CLR PX.Y (e.g., CLR P1.2), and SETB PX.Y (e.g., SETB P2.0). These instructions are directed to the latches rather than to ports in order to avoid misinterpretation of the pin voltage level. For instance, a one bit of a port can be used to supply voltage to a base of a bipolar transistor. When a logical level equal to 1 is written to the bit, the transistor is turned on. If the microprocessor reads the same bit from the port through its physical pin instead of the latch output, it reads the voltage of the base of the transistor, and thus it interprets as a logical level equal to 0. By reading the output of a latch instead of a pin, the logic level that will be returned is the logic level equal to 1 [1–10].

A bit of a port is programmed as input if a write operation of a logical level equal to 1 on the latch of the port is performed. As we have already seen, consider Fig. 6.3, when the output Q of the latch presents a logic level equal to 1, the output \bar{Q} presents a logic level equal to 0. When this occurs, the transistor operates as an open key, the pin information can be captured by the microprocessor through the data bus, and the contents of the ports can read information of external input interfaces [1–10].

The external input interfaces commonly used with the microcontrolled systems use transducers (electronics circuits with sensors) with the purpose of controlling the variables of a process to be managed. Examples of transducers used as external input interfaces are those related with the temperature, pressure, humidity, optical, magnetic, capacitive, inductive, etc. Other examples of external input interfaces commonly used as external information inputs for microcontrolled systems are interfaces with keyboard, push-button switches, dipswitches, mouse, scanners, etc. [1–10].

6.3 Some Examples of Programming in Assembly Using the Input/Output Ports

Some examples of programming of input/output ports are presented in this Section. Each bit of the ports can operate either as data input or as data output. When they are configured as data input, the logic levels are defined by the external interfaces to the microcontrolled system, such as the dipswitches, sensor (temperature, pressure, etc.), and transducers. When they are configured as data output, the logic levels are defined by the microcontrolled system to the external interfaces (LEDs, DC motor, lamps, etc.) [1–10].

Example 1 Programming all bits of the ports P0 to work as inputs, which are externally defined by the external interfaces for the microcontroller system:

MOV P0, #0FFh

The symbolic representation of this instruction is $(P0) \leftarrow \#0FFh$. It means that the content of port 0 (P0) will be initialized with the constant value FFh. This instruction writes logic 1 into the eight latches of P0, and therefore all the bits of P0 are programmed to operate as input bits of external data to the microcontrolled systems.

Note This instruction requires that you always add a zero before the hexadecimal numbers that begin with letters (Ah to Fh). If this is not done, it acknowledges an error in the compilation process, indicating that this value corresponds to a variable that was defined by the programmer.

Example 2 Programming the most significant bit of P0 to work as an input to the microcontroller and the rest of the bits are programmed as data output from the microcontroller system to the external world:

MOV P1, #80h

The symbolic representation is $(P1) \leftarrow \#80h = \#10000000b$ (b means binary representation), indicating that the content of P1 will be initialized with the constant value 80h. This instruction writes logic 1 into the most significant bit of the latch of P1 and logic 0 in the rest of other bits of the latch of P1. The most significant bit of P1 is programmed as input and the others as output.

Example 3 Programming the even bits (0, 2, 4, and 6) of P0 as data input of the microcontrolled system and the odd bits (1, 3, 5, and 7) of P0 as data output of the microcontroller system to the external world.

MOV P2, # 55h

Its symbolic representation is $(P2) \leftarrow \#55h = \#01010101b$, meaning that the content of P2 will be initialized with the constant value 55h. This instruction writes logic 1 in the bits 0, 2, 4, and 6 of the latch of P2 and logic 0 in the bits 1, 3, 5, and 7 of the latches of P2. The even bits of P2 will be programmed as inputs and the odd bits as outputs.

Example 4 To illustrate some read operations by using the family of 805 core microcontrollers, firstly it is necessary to program the bits of ports to operate as input. Once programmed, it is possible to do the data reading operations through them. It is important to observe that after a reset signal, all output of latches of the ports are defined by hardware with high logic level, and therefore all bits of the ports are programmed as inputs. The instructions MOV A, PX (X = 0–3), MOV Rn, PX (X = 0–3), MOV @ Ri, and PX (X = 0–3) are examples of instructions that read the contents of ports. Consider the instructions:

Mnemonic	Argument	Symbolic representation	Comments
MOV	A, #0FFh	$(A) \leftarrow \#0FFh$; It initializes the content of the accumulator ; (A) register with the ; value FFh
MOV	P0, A	$(P0) \leftarrow (A)$; It programs all bits of P0 as inputs, because the ; content of the ; accumulator is equal to FFh and when we do a ; write operation of

(continued)

			; logic 1 in the output of the latch of the port, it is ; programmed ; as input
MOV	A, P0	(A)←(P0)	; It performs a read operation of the content of P0 ; to the content of the ; accumulator (A) register. P0 contains data that ; are defined ; externally by an input interface. This ; information may have been ; defined by an interface formed by a dipswitch ; that was connected ; to port P0, for example
END			

Example 5 In the same way that it had been done, before doing a write operation in the ports, it is necessary to do the programming of each port. The instructions MOV PX, A; MOV PX, Rn; MOV PX, @ Ri; MOV PX, #data, with (X = 0–7); and ANL, ORL, and XRA instructions with the different addressing modes are also some other examples of instructions which can do write operations in the ports. To illustrate, consider the set of instructions described below:

Mnemonic	Argument	Symbolic representation	Comments
MOV	A, #00h	(A)←#00h	; It initializes the content of the accumulator ; (A) register with the ; value 00h
MOV	P0, A	(P0)←(A)	; It configures all bits of P0 as outputs, because ; the content of the ; accumulator is equal to 00h and when we do a ; write operation of ; logic 0 in the output of the latch of the port, it is ; programmed ; as output
MOV	P0, R5	(P0)←(R5)	; It performs a write operation of the content of ; the R5 register in ; the content of P0. Depending on the value of ; the content of R5 ; register, the content of the P0 bits can be ; defined either with logic ; 0 or 1. If this port is electrically connected with ; LEDs taking into ; account positive logic, for example, the bits ; with logic 1 will ; activate the LEDs and the bits with logic 0 will ; deactivate the LEDs
END			

Example 6 To monitor the digital signals generated by the input interfaces, for instance, those implemented with keys or senses, they must use the instructions:

- I. JB PX.Y, addr1, where X represents the ports (P0, P1, P2, and P3), Y represents the bits of the ports (bit 0, bit 1, ..., bit 7), and addr1 is the address to which the microprocessor will jump to run other instructions: If the content of bit Y of port X (PX.Y) is equal to 1, then this instruction jumps to address addr1 [(PC) \leftarrow addr1].

Note If this instruction is written as JB PX.Y, \$, it means that if the content of bit Y of port X (PX.Y) is equal to 1, then this instruction jumps to its own address of program memory [(PC) \leftarrow \$.]. The objective of this structure is to simplify the way to write this instruction instead of writing this instruction regarding a traditional method, as indicated below:

addr1: JB PX.Y, addr1

- II. JNB PX.Y, addr1, where X represent the ports (P0, P1, P2, and P3), Y represents the bits of the ports (bit 0, bit 1, ..., bit 7), and addr1 is the address to which the microprocessor will jump to run other instructions: If the content of bit Y of port X (PX.Y) is equal to 0, then this instruction jumps to address addr1 [(PC) \leftarrow addr1].

Note If this instruction is written as JNB PX.Y, \$, it means that if the content of bit Y of port X (PX.Y) is equal to 0, then this instruction jumps to its own address of program memory [(PC) \leftarrow \$.]. The objective of this structure is to simplify the way of writing this instruction instead of doing so regarding a traditional method, as indicated below:

addr1: JNB PX.Y, addr1

Therefore, in order to detect the transition from logic 1 to logic 0 (falling edge) of a single bit of a port, you can develop a program in Assembly as follows:

Address of the program memory	Mnemonic	Argument	Symbolic representation	Comments
	JB	P1.0, \$	(PC) \leftarrow (PC) + 3	; It calculates the address of the next instruction to be run, i.e.; three units are added to the contents of the program counter (PC) register, as the instruction JB P1.0, \$; consists of three bytes.
			If (P1.0) = 1 \Rightarrow (PC) \leftarrow \$; It means jumping to the address of the instruction itself, while the content of bit 0 of port 1 (P1.0) is equal to logic 1, i.e. the program jumps to the address of the instruction JB

(continued)

				; P1.0, \$. This looping occurs ; until the content of bit 0 of ; P1 changes to logic 0. When ; the content of bit 0 of port P1 ; changes from logic 1 to logic ; 0, the program will ; run the address immediately ; following this instruction, ; which ; corresponds to the address of ; the next instruction to be ; run
	END			

Example 7 In order to detect the transition from logic 0 to logic 1 (rising edge) of a single bit of a port, you can develop a program in Assembly as follows:

Address of the program memory	Mnemonic	Argument	Symbolic representation	Comments
	JNB	P1.0, \$	(PC) ← (PC) + 3	; It calculates the address of the ; next instruction to be run, i.e. ; three units are added to the ; contents of the program ; counter (PC) register, as the ; instruction JNB P1.0, \$; consists of three bytes.
			if (P1.0) = 0 ⇒ (PC) ← \$; It means jumping to the ; address of the instruction ; itself, while ; the content of bit 0 of port ; 1 (P1.0) is equal to logic 0, ; i.e. the program jumps to the ; address of the instruction JNB ; P1.0, \$. This looping occurs ; until the content of bit 0 of ; P1 changes to logic 1. When ; the content of bit 0 of port P1 ; changes from logic 0 to logic ; 1, the program will ; run the address immediately ; following this instruction, ; which ; corresponds to the address of ; the next instruction to be ; run
	END			

Example 8 It is possible to manage a combination of activations defined by the input ports by using the *CJNE A, direct, addr* instruction. This instruction is able to calculate the address of the next instruction by using this $[(PC) \leftarrow (PC) + 3]$. Afterwards, if the content of the accumulator (A) register is different from the content of the memory whose address is “*direct*,” it jumps to the address defined as “*addr*” and defines the carry-bit flag indicating if the (A) is smaller, higher, or equal to the content of the memory whose address is “*direct*” [if $(A) \neq (\text{direct})$ then $(PC) \leftarrow \text{addr}$; if $(A) < (\text{direct})$, then $(C) \leftarrow \#1b$ otherwise $(C) \leftarrow \#0b$].

For this, we must define a standard value (byte) in the content of the accumulator (A) register and compare it with the content of the memory whose address is “*direct*.” This standard value corresponds to a binary combination of input bits that you want to manage, detect, monitor, or control. If the content of the accumulator (A) register is different (\neq) from the content of the port, the program must jump to the address of the instruction itself (\$), i.e., the program jumps continuously to the own address of this instruction (it stays in loop). When the content of the port is set externally with a value which is equal to the accumulator (A) register (standard value desired), the program runs to the address immediately following this instruction (*CJNE A, direct, addr*).

To illustrate, consider that port P1 is electrically connected to an external input interface consisting of a dipswitch with eight keys (negative logic: when the content of a bit of the port is defined with logic 1, the key is deactivated and when the content of a bit of the port is defined with logic 0, the key is activated), and initially all bits of port P1 are set to logic 1 (all keys are deactivated). Therefore, the program below stays waiting until the content of bits 0 and 7 of P1 becomes equal to logic 0 (keys 7 and 0 be activated) and the rest stay equal to 1 (keys 6, 5, 4, 3, 2, and 1 deactivated).

Address of the program memory	Mnemonic	Argument	Symbolic representation	Comments
	MOV	A, #7Eh	$(A) \leftarrow \#01111110b$; It defines the input to be detected, i.e. the least and most significant bits must be activated and the others must not be activated (positive logic);
	CJNE	A, P1, \$	If $(A) \neq (P1) \Rightarrow (PC) \leftarrow \$$ If $(A) < (\text{direct}) \Rightarrow$; It compares (A) to (P1), i.e. it jumps to the same ; CJNE instruction address if they

(continued)

			(C)←#1 else (C)←#0	; are different. This instruction ; waits for the content of port ; P1 to become equal to the ; content of accumulator (A), ; i.e. it stays in loop while this ; situation occurs. When ; (A)=(P1), the program exits ; this ; loop and executes the ; immediately following ; instruction ; to the CJNE instruction, it ; means it detected the ; triggering.
	END			

Example 9 To detect the activation of any bit of a port, the comparison process can be used, for instance, by means of the subtraction instruction. Therefore, consider that P1 is electrically connected to an eight-key dipswitch (negative logic) and when it is deactivated, logic 1 is defined on all bits of P1.

Address of the program memory	Mnemonic	Argument	Symbolic representation	Comments
WAIT:	MOV	A, #0FFh	(A)←#11111111b	; It defines a binary combination that considers all bits equal to one, i.e. it considers all keys to be disabled/deactivated
	CLR	C	(C)←#0 ₂	; It resets the carry-bit flag so that it does not interfere with the result of the subtraction operation.
	SUBB	A, P1	(A)← (A) + [(C) + (P1)] _{C2} (C) ← (C̄) and (AC) ← (AC̄)	; It subtracts the contents of the carry-bit flag and port 1 from the content of the accumulator (A) register.
	JZ	WAIT	Se (Z) = 1 [(A)=0]⇒(PC)←WAIT	; If no key has been activated, it means that the content of the accumulator (A) register is equal to the content of port 1, i.e. no key was activated. ; Therefore, the zero-flag is set (changes to logic 1) and the program jumps to the address WAIT (it stays in loop)

(continued)

				; until at least one key is ; activated. So, when ; the content of the accumulator ; (A) register becomes ; different from the content of ; port 1, the zero-flag ; becomes equal to zero and ; therefore the program ; runs the instruction after the ; instruction JZ WAIT.
	END			

Another possible solution to this problem is given below.

Address of the program memory	Mnemonic	Argument	Symbolic representation	Comments
WAIT:	MOV	A, P1	(A) ← (P1)	; It considers initially that all the bits ; of port 1 are ; equal to 1 [(P1)=FFh] (no key is ; activated, i.e. ; no key is in logical 0, because we ; are ; considering negative logic)
	INC	A	(A) ← (A) + 1	; If no key has been activated, the ; content of the ; accumulator (A) register is added ; to one unit ; becoming equal to zero (FFh ; +1=00h). ; Consequently, the zero-flag is set ; (becomes equal ; to 1). However, if any key is ; activated after the ; addition operation, the content of ; the ; accumulator (A) register becomes ; different from ; zero, and consequently the zero- ; flag becomes ; equal to zero (Z=0)
	JZ	WAIT	Se (Z) = 1 [(A)=0] ⇒ (PC) ← WAIT	; It waits for the activation of any ; key of the port while the content of ; the accumulator (A) register is ; equal to 0, as no key was activated. ; If any key is activated, the program ; runs the instruction after the JZ ; instruction.
	END			

6.4 Routines to Generate Time (Timing Routines)

It is common in computer systems to define certain conditions of operation based on the time, such as the time to activate/deactivate a LED, the time to update the information of a display, the time to activate a buzzer/beep, the time to turn on/off a machine, etc. [10].

There are two ways to generate time in computer systems using microcontrollers: through hardware, using the Timers/Counters and through software, and using a set of instructions specially arranged in a program (software) [10].

To generate time through software, firstly it is necessary to choose a particular register or a certain data memory location. Then, it is necessary to initialize its content with a certain value and finally decrement its content until its value becomes equal to zero. An example of this type of routine is shown in Fig. 6.4 [10] (Fig. 6.6).

Observing the instruction set of the 805 core microcontroller family (Appendix A), the needed time to execute this program is calculated as follows:

Instruction	Number of times that the instruction is run	Quantity of clock cycles to run this instruction	Total clock cycles per instruction
MOV R0, #constant	1	1	1.1 = 1
DJNZ R0, \$	Constant (it can vary from 00h to FFh)	2	2.constant

Therefore, the number of clock cycles or machine cycles (CM) required to perform this routine is given by Eq. (6.1):

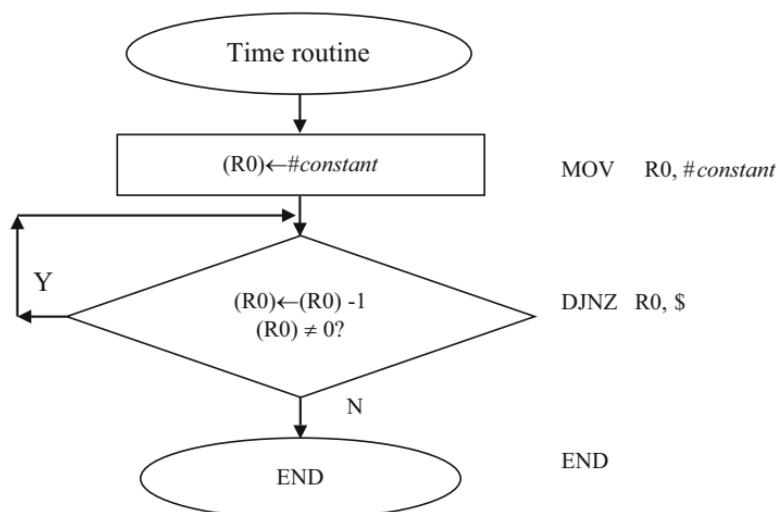


Fig. 6.6 Flowchart and source program in Assembly of a routine that generates time through software using a single register (counting down)

$$CM = 1.1 + \text{constant.}2 = 1 + 2.\text{constant} \quad (\text{clock cycles}) \quad (6.1)$$

The frequency of a clock cycle ($f_{\text{clock_cycles}}$) of the 8051 core microcontroller family corresponds to $1/12$ of the crystal frequency (f_{crystal}), as indicated in Eq. (6.2):

$$f_{\text{clock_cycles}} = (1/12).f_{\text{crystal}}(\text{Hz}) \quad (6.2)$$

The period of a clock cycle ($T_{\text{clock_cycles}}$) is the inverse of the frequency of a clock cycle, according to Eq. (6.3):

$$T_{\text{clock_cycles}} = 1/f_{\text{clock_cycles}} = 12/f_{\text{crystal}}(\text{s}) \quad (6.3)$$

Thus, the total time (T_{total}) generated by this routine is given by Eq. (6.4):

$$T_{\text{total}} = CM \cdot T_{\text{clock_cycles}} = (1 + 2.\text{constant}).12/f_{\text{crystal}}(\text{s}) \quad (6.4)$$

To illustrate, consider that a microcontroller is using a crystal (f_{crystal}) which is equal to 12 MHz, and the value of the “*constant*” is equal to 256. Then, the total time (T_{total}) of that time routine, applying Eq. (6.4), is:

$$\begin{aligned} T_{\text{total}} &= (1 + 2.\text{constant}).12/f_{\text{crystal}} = (1 + 2.256).12/12.10^{-6} = 513.10^{-6} \\ &= 513 \mu\text{s} = 0.513 \text{ ms} \end{aligned}$$

Observe that the time of the routine that uses a single register or memory location is quite small considering the perception of the humans [10].

To generate larger values of time, an example is presented by the flowchart of Fig. 6.7, which uses two registers (R0 and R1). These registers will have the function of being two counters which are decreasing. The technique used to generate this greater time consists of putting a Counter inside another counter: whenever the first Counter (R0) is decremented by one unit, the second Counter will be decremented several times, depending on its initial value, i.e., for each decrement of R0 register of one unit, the content of the R1 register will be decremented by one unit “*constant2*” times [10].

Observing the instruction set of the 8051 core microcontroller family (Appendix A), the needed time to execute this program is calculated as follows:

Memory program address	Instruction	Number of times that the instruction is run	Quantity of clock cycles to run this instruction	Total clock cycles per instruction
	MOV R0, #constant1	1	1	1.1 = 1
addr1:	MOV R1, #constant2	constant1	1	1.constant1
	DJNZ R1, \$	constant1.constant2	2	2.constant1.constant2
	DJNZ R0, addr1	constant1	2	2.constant1
	END			

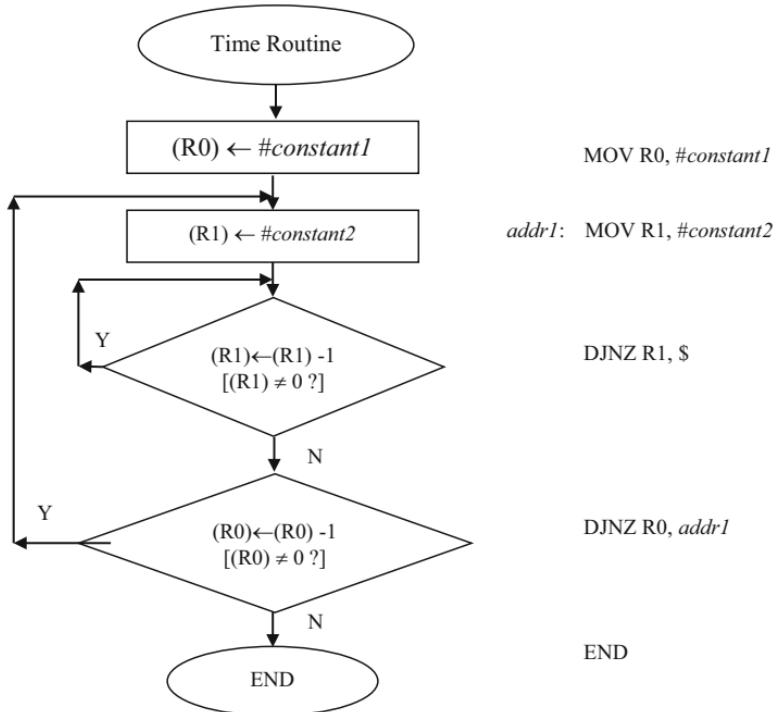


Fig. 6.7 Flowchart and source program in Assembly of a routine that generates time through software using two registers (counting down)

Therefore, the number of clock cycles or machine cycles (CM) required to perform this routine is given by Eq. (6.5):

$$CM = 1 + 3.\text{constant1} + 2.\text{constant1}.\text{constant2} \text{ (cycles)} \quad (6.5)$$

Analogously to the calculations developed before, the total time (T_{total}) generated by this routine is given by Eq. 6.6:

$$T_{\text{total}} = (1 + 3.\text{constant1} + 2.\text{constant1}.\text{constant2}).12/f_{\text{crystal}}(\text{s}) \quad (6.6)$$

To illustrate, the maximum total time that this routine is able to generate can be calculated by applying Eq. 6.6, considering f_{crystal} equal to 12 MHz, where the values of the R0 and R1 registers must be equal to the value 256 (in this routine, the content of the R0 and R1 registers must be initialized with the value 00h, since it first decrements by one unit the registers and then compares them with zero), we have

$$T_{\text{total}} = 12.(1 + 3.256 + 2.256.256)/(12M) = 131,841 \mu\text{s} \approx 132 \text{ ms} \approx 0.132 \text{ s}$$

to generate even longer times, you must use more counters (three, four, etc.) by using the same approach (one Counter inside another).

6.5 Monitoring Software for Mechanical Keys

The mechanical key is still an important input device for computer systems, as it is a basic component of keyboards, dipswitches, push buttons, etc. However, the mechanical keys present an effect called “bounce” when they are turned on (from logic level 1 to logic level 0) and turned off (from logic level 0 to logic level 1), as shown in Fig. 6.8 [10].

In Fig. 6.8, V_{DD} is the supply voltage, T_{bounce} is the period of the bounce (typical values: from units of microseconds to cents of milliseconds, depending on the manufacture), and T is the time necessary to perform another read of the bit of the port which is electrically connected to an electrical interface of the key to verify if it was really activated. T must be higher than T_{bounce} to guarantee the read of the logic level of the key correctly due to its bounce [10].

The bounce (electrical noise) generated by a mechanical key strongly affects the performance of a routine (software) that monitors its activation and deactivation. The strategy to design a piece of software to detect the activation of a key, eliminating the bounce, is [10]:

- I. Firstly, we must check the electrical condition of the key over time (it can be checked continuously or in intervals of time).
- II. While the logic level is equal to 1 (the key is turned off), we must wait for its activation to occur.
- III. When it is activated, i.e., when the logic level of the interface circuit output goes from 1 to 0 (point 1 in Fig. 6.8), we must wait for the extinction of the key bounce, i.e., waiting for a time (T) higher than T_{bounce} to perform another read of the bit of the port which is electrically connected to the key interface. If it was really activated (point 2 in Fig. 6.8), then we can confirm that the key was activated and run other instructions which are dependent on this activation, otherwise the key was not really activated (point 3 in Fig. 6.8), and therefore the process to detect the key activation must be reinitialized (return to the beginning of this routine).

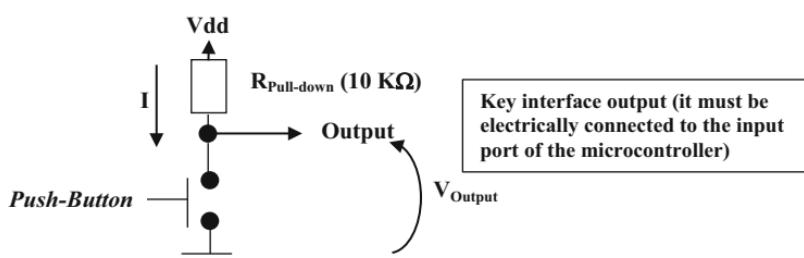


Fig. 6.8 Input interface using a mechanical push-button key and its output electrical signal emphasizing the bounce (electrical noise) generated when it is turned on and turned off

Another way of eliminating the bounce of a mechanical key is through the use of additional hardware, e.g., using a monostable multivibrator (e.g., 555) or NAND ports configured as SC-type flip-flops which must be connected to the output pin of the key interface. This strategy increases the cost of the hardware [10].

6.6 Examples of Mechanical Key Monitoring Routines

This section presents some examples of monitoring routines for mechanical keys.

Example 1 Considering that an interface for dipswitches with eight keys is electrically connected to the port 1 (P1: configured as input), a software example to monitor the activation of any mechanical key, considering the bounces of the keys, is shown in Fig. 6.9.

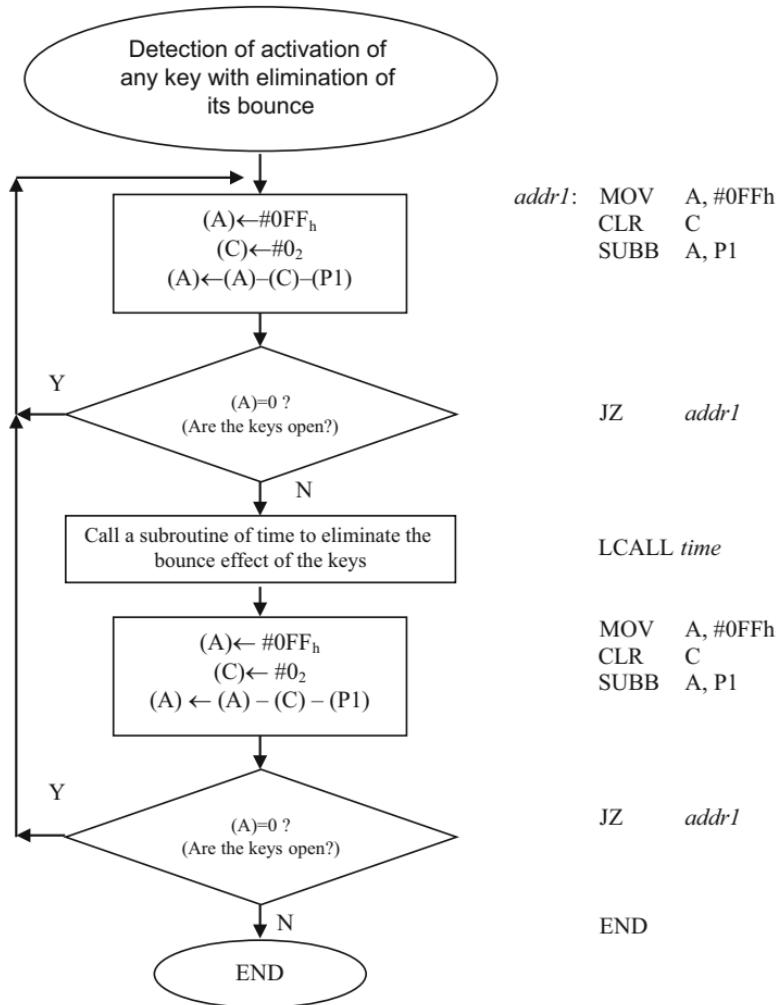


Fig. 6.9 Flowchart and source program of a routine that detects the activation of any mechanical key with bounce elimination

In Fig. 6.9, *time* in source program means the address of a subroutine that generates time (it can be composed of one register, two registers, etc.).

The source program corresponding to Fig. 6.9 is presented below, including the time routine. Constant1 and constant2 can assume values from 00h to FFh.

Address of the memory program	Mnemonic	Argument	Comments
WAIT:	MOV	A, #0FFh	; It is waiting for at least one key to be activated
	CLR	C	
	SUBB	A, P1	
	JZ	WAIT	
DELAY:	MOV	R0, #constant1	; Time routine regarding two registers to ; generate a time (T) higher than T_{bounce} to ; eliminate the bounce effects of the keys during ; the detection of their activations
	MOV	R1, #constant2	
	DJNZ	R1, \$	
	DJNZ	R0, DELAY	
	MOV	A, #0FFh	; It confirms the activation of at least one key
	CLR	C	
	SUBB	A, P1	
	JZ	WAIT	; If an activation of a key is not confirmed, the ; process of detection of at least one key restarts
		END	; Activation of at least a key is confirmed

Example 2 Considering that an interface for dipswitches with eight keys is electrically connected to the port 1 (P1: configured as input), a software example to monitor the deactivation of all mechanical keys, considering the bounces of the keys, is shown in Fig. 6.10.

Address of the memory program	Mnemonic	Argument	Comments
WAIT:	MOV	A, #0FFh	; It is waiting for the deactivation of all keys
	CLR	C	
	SUBB	A, P1	
	JNZ	WAIT	
	MOV	R0, #constant1	; Time routine regarding two registers to ; generate a time (T) higher than T_{bounce} to ; eliminate the bounce effects of the keys during ; the detection of their deactivations

(continued)

DELAY:	MOV	R1, #constante2	
	DJNZ	R1, \$	
	DJNZ	R0, DELAY	
	MOV	A, #0FFh	; It confirms the deactivation of all keys
	CLR	C	
	SUBB	A, P1	
	JNZ	WAIT	; If the deactivation of all keys is not confirmed, ; the process of detection of all keys restarts
	END		; Deactivation of all keys is confirmed

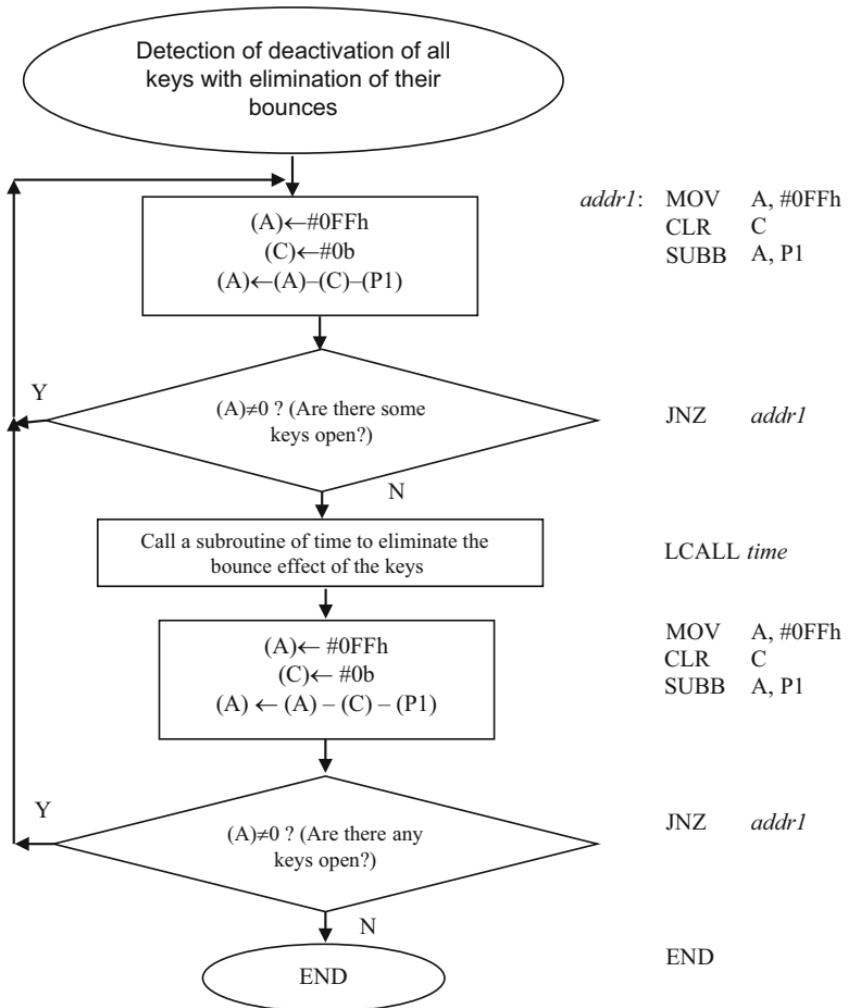


Fig. 6.10 Flowchart and source program of a routine that detect the deactivation of all mechanical keys with bounce elimination

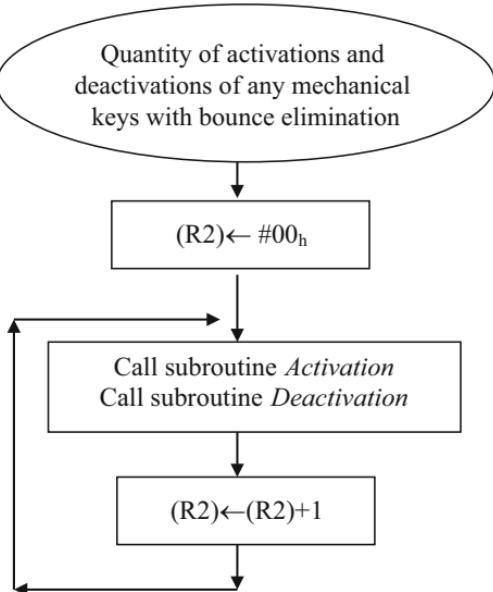


Fig. 6.11 Flowchart and source program that counts the quantity of activation and deactivation of any mechanical keys, with bounce elimination

Example 3 Considering that an interface for eight-key dipswitches is electrically connected to the port 1 (P1: configured as input), Fig. 6.11 presents a structured software in Assembly (flowchart and source program) using a member of the 8051 core microcontroller family that counts the quantity of activations and deactivations of any mechanical keys, with elimination of the bounce. Observe that two subroutines are used: the first subroutine monitors the activation process, and the second subroutine monitors the deactivation process of the keys. Each time that at least one key is activated and subsequently all the keys return to the initial deactivated condition, the content of the R2 register (which is initially zero) is incremented by one unit.

The source program that is not structured (without using the subroutines) regarding Example 3, which counts the number of activations and deactivations of keys with bounce elimination, is presented below.

Address of the memory program	Mnemonic	Argument	Comments
	MOV	R2, #00h	
WAIT1:	MOV	A, #0FFh	; It waits for the activation of at least one key
	CLR	C	
	SUBB	A, P1	
	JZ	WAIT1	
	MOV	R0, #constant1	; Time routine to eliminate the bounce

(continued)

DELAY1:	MOV	R1, #constant2	
	DJNZ	R1, \$	
	DJNZ	R0, DELAY1	
	MOV	A, #0FFh	; Confirmation of the activation
	CLR	C	
	SUBB	A, P1	
	JZ	WAIT1	
WAIT2:	MOV	A, #0FFh	; It waits for the deactivation of all keys
	CLR	C	
	SUBB	A, P1	
	JNZ	WAIT2	
	MOV	R0, #constante1	; Time routine to eliminate the bounce
DELAY2:	MOV	R1, #constante2	
	DJNZ	R1, \$	
	DJNZ	R0, DELAY2	
	MOV	A, #0FFh	; Confirmation of the deactivation of all keys
	CLR	C	
	SUBB	A, P1	
	JNZ	WAIT2	
	INC	R2	; It counts the quantity of times the process ; of activation and ; deactivation of the keys occurs
	SJMP	WAIT1	

6.7 Resolved Exercises

6.7.1. Design the following flowcharts and structured source programs in Assembly using one of the members of the 8051 core microcontroller family, considering that the port 0 is electrically connected to an input interface with an eight-key dipswitch (1, open key; 0, closed key) and the port 1 is electrically connected to a set of eight LEDs (positive logic: logic 0 deactivates LED and logic 1 activates LED), considering the following:

I. This software must show a binary down Counter on the LEDs for a period of 0.6 s whenever key 2 (which is electrically connected to bit 2 of port 0) is activated (consider bounce).

Solution Firstly, we must configure the hardware and define a register (or a memory location) to be the counter. The subroutine below configures the P0 and P1 as input and output, respectively. Besides, we must define the content of the R7 register to be a binary down counter, according to Fig. 6.12.

Regarding the bounce, we can use a single down counter, as presented in Fig. 6.6, considering constant1 equal to 00h (maximum value, as this routine firstly decrements by one unit and after it verifies if the result is different from zero, which corresponds to a maximum time of 0.5 ms). In this software, it is called “Tiboun,” according to Fig. 6.13. The maximum time that the time routine with two counters generates according to Eq. (6.6) is approximately 0.13 s. Therefore, to generate 0.6 s, we must implement another routine with three counters, according to Fig. 6.14.

Fig. 6.12 Flowchart and source program that configure the ports and define the down binary counter

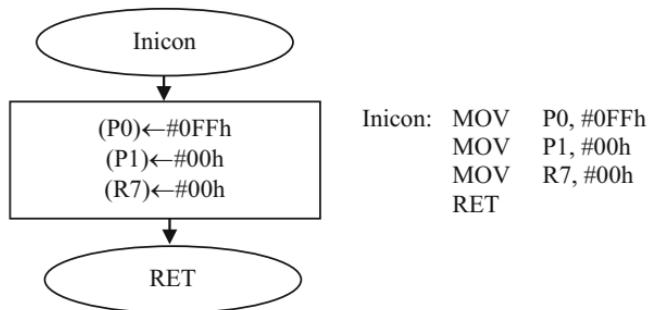
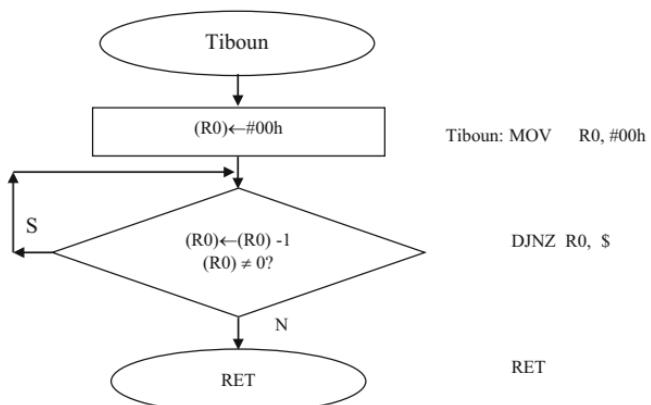


Fig. 6.13 Flowchart and source program in Assembly of a routine that generate time of 0.5 ms via software by using a single register (counting down)



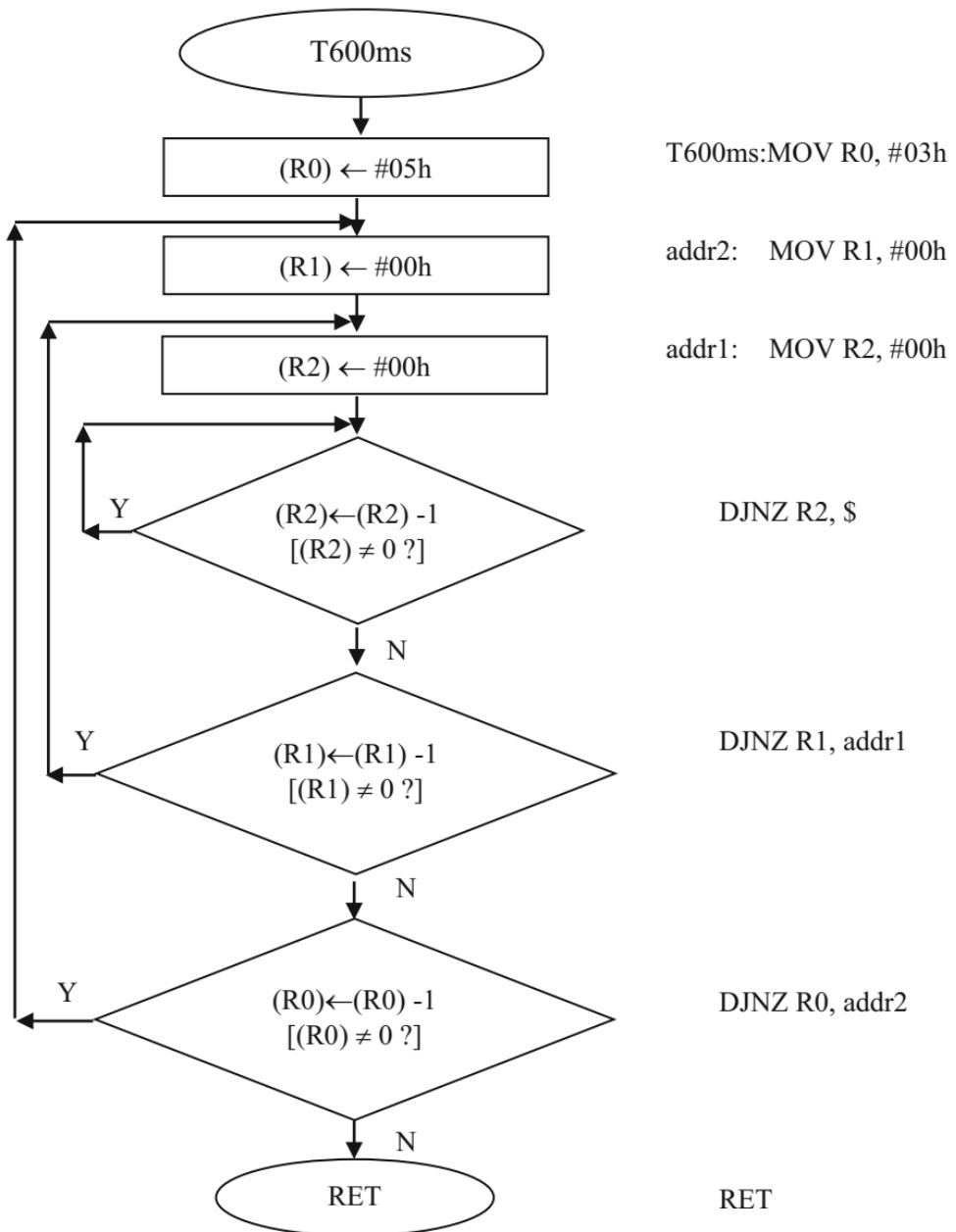


Fig. 6.14 Flowchart and source program in Assembly of a routine that generate time via software by using three registers (counting down)

Hence, observing the instruction set of the 8051 core microcontroller family (Appendix A), the time taken to execute this program is calculated as follows:

Memory program address	Instruction	Number of times that the instruction is run	Quantity of clock cycles to run this instruction	Total clock cycles per instruction
T600 ms:	MOV R0, #constant1	1	1	1.1 = 2
addr2:	MOV R1, #constant2	constant1	1	1.constant1
addr1:	MOV R2, #constant3	constant1.constant2	1	1. constant1.constant2
	DJNZ R2, \$	constant1.constant2.constant3	2	2. constant1.constant2.constant3
	DJNZ R1, addr1	constant1.constant2	2	2. constant1.constant2
	DJNZ R0, addr2	constant1	2	2.constant1
	END			

Therefore, the number of clock cycles or machine cycles (CM) required to perform this routine is given by Eq. (6.7):

$$CM = 1 + 3.\text{constant1} + 3.\text{constant1}.\text{constant2} + 2.\text{constant1}.\text{constant2}.\text{constant3} \text{ (cycles)} \quad (6.7)$$

Analogously to the calculations developed before, the total time (T_{total}) generated by this routine is given by Eq. (6.8):

$$\begin{aligned} T_{\text{total}} &= (1 + 3.\text{constant1} + 3.\text{constant1}.\text{constant2} + 2.\text{constant1}.\text{constant2}.\text{constant3}) \cdot \\ &\quad 12/f_{\text{crystal}}(\text{s}) \end{aligned} \quad (6.8)$$

To illustrate, the maximum total time that this routine is able to generate can be calculated by applying Eq. 6.8, considering f_{crystal} equal to 12 MHz, where the values of the R0, R1, and R2 registers must be equal to the value 256 (in this routine, the content of the R0, R1, and R2 registers must be initialized with the value 00h, since it first decrements by one unit the registers and then compares them with zero), we have

$$T_{\text{total}} = 12.(1 + 3.256 + 3.256.256 + 2.256.256.256)/(12M) = 33.8 \text{ s}$$

Then, with three counters it is possible to generate 0.6 s because it can generate up to 50.7 s. Thus, in order to determine the values of the contents of R0, R1, and R2, we can define the values of two of them [e.g., (R1) and (R2)] with the value 256 and determine the other (R0) considering T_{total} of Eq. (6.8) equal to 0.6 s, as follows:

$$\begin{aligned} T_{\text{total}} &= 12.(1 + 3.\text{constant1} + 3.\text{constant1}.256 + 2.\text{constant1}.256.256)/(12M) \\ &= 0.6 \text{ s} \end{aligned}$$

Therefore, the value of constant1 that must be defined in the content of the R0 register is equal to 4.6 and therefore approximately 5.

The flowchart and the source program of the subroutine that are able to detect the activation of key 2 considering the effects of the bounce are shown in Fig. 6.15.

Finally, the flowchart of the main program is illustrated in Fig. 6.16.

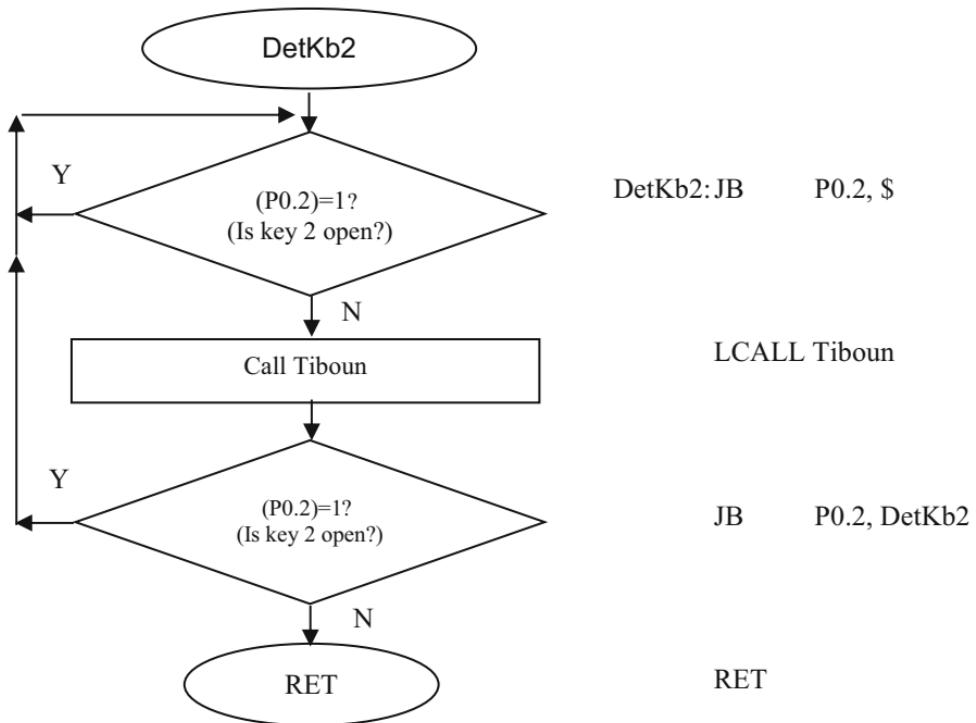
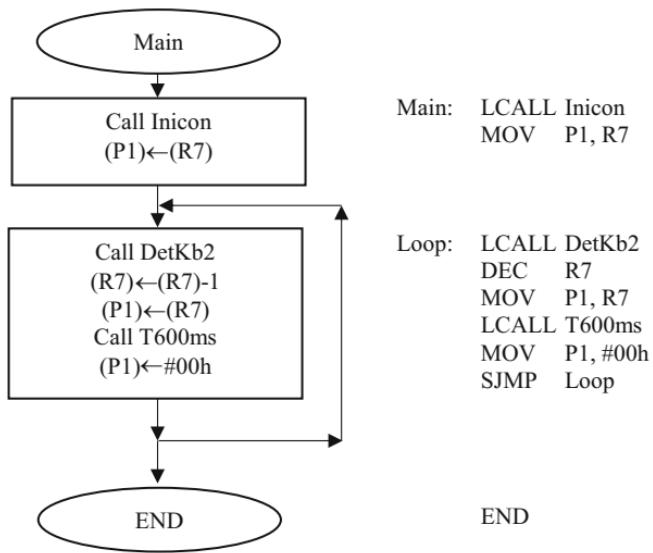


Fig. 6.15 Flowchart and source program in Assembly of a subroutine that are able to detect the activation of key 2 considering the bounce)

Fig. 6.16 Flowchart and source program in Assembly of the main program regarding the exercise 6.7.1 – I



The source program of this example is described below:

Address of the memory program	Mnemonic	Argument	Comments
Inicon:	MOV	P0, #OFFh	; Subroutine of the ports' configuration and ; initialization ; of the down counter (R7)
	MOV	P1, #00h	; It configures (P1) as output
	MOV	R7, #00h	; It initializes the down counter defined by ; (R7) (256 ; down counting)
	RET		; End of subroutine Inicon
Tiboun:	MOV	R0, #00h	; Subroutine of the time to eliminate the bounce ; of the ; key
	DJNZ	R0, \$; It initializes the down counter defined by ; (R0) (256 ; down counting generating a time of 0.5 ms) to ; be used ; to eliminate the bounce of the key
	RET		; End of subroutine Tiboun
T600 ms:	MOV	R0, #05h	; Subroutine of the time to generate 0.6 s (600ms)
addr2:	MOV	R1, #00h	
addr1:	MOV	R2, #00h	

(continued)

	DJNZ	R2, \$	
	DJNZ	R1, addr1	
	DJNZ	R0, addr2	
	RET		; End of subroutine T600ms
			; Subroutine to detect the activation of key 2 ; regarding its bounce
DetKb2:	JB	P0.2, \$; It waits for the activation of key 2. If the content ; of ; bit 2 of port 0, which has the logic condition of ; the ; key 2 condition (open or closed), is equal to ; 1 (it is ; open) then this instruction jumps to itself ; (DetKb2 ; address)
	LCALL	Tiboun	; If key 2 is closed (bit 2 goes to logic 0), ; subroutine ; Tiboun is called in order to avoid the effect of ; the bounce of ; key 2 regarding the detection of its activation
	JB	P0.2, DetKb2	; It confirms if key 2 was really activated. If the ; content ; of bit 2 of port 0, which has the logic condition ; of the ; key 2 condition (open or closed), is equal to ; 1 (it is ; open) then this instruction jumps to the DetKb2 ; address (the activation of key 2 was not ; confirmed and ; thus the process to detect key 2 must be ; reinitialized)
	RET		; If the key was really activated [the content of bit ; 2 of (P0) goes to logic 0], then this subroutine is ; ended
			; Main program
Main:	LCALL	Inicon	; Call subroutine Inicon
	MOV	P1, R7	; It writes in the content of P1, which is ; electrically ; connected to the LEDs with the initial value of ; (R7)
Loop:	LCALL	DetKb2	; It calls subroutine DetKb2 (detect the activation ; of key ; 2 with elimination of the bounce)
	DEC	R7	; If key 2 was activated, (R7) is decremented by ; one unit
	MOV	P1, R7	; It update the information on the LEDs
	LCALL	T600ms	; Call T600ms
	MOV	P1, #00h	; Clear P1
	SJMP	Loop	; It jumps to the loop of the main program
	END		

II. This software must show a decimal down Counter on the LEDs for a period of 0.6 s whenever key 2 (which is electrically connected to bit 2 of port 0) is activated (consider bounce).

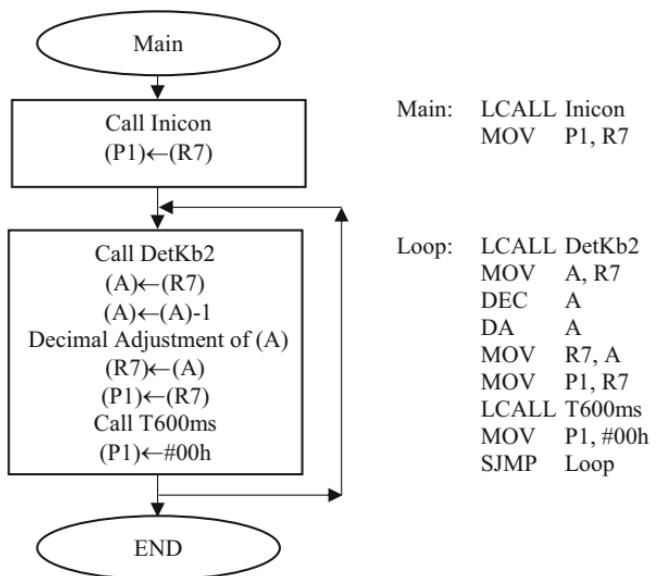
Solution The only change in the software 6.7.1 – I described above is in the main program. We must move the content of the R7 register to the accumulator (A) register, decrement the content of the accumulator (A) register by one unit, perform a decimal adjustment by using the content of the accumulator (A) register, and store the result in the content of the (R7) register, as indicated in Fig. 6.17.

III. This software must show a binary down Counter on the LEDs for a period of 0.6 s whenever keys 3 and 6 (which are electrically connected to bit 3 and bit 6, respectively, of port 0) are activated (consider bounce).

Solution The only change in the software 6.7.1 – I described above is in the subroutine that detects the activation of key 2. Therefore, we must carry out a masking operation with the content of the P0 register. The masking operation consists of performing an AND logic operation of the content of the P0 register with a constant. This constant is calculated by defining 1 to the keys of which we want to detect the activations and 0 to the keys of which we do not want to detect the activations. So, we must define 0 to the bits 0, 1, 2, 4, 5, and 7, as indicated in Fig. 6.18.

Observe that when we perform an AND logic operation of a bit with logic 0, the result is 0, and when we perform an AND logic operation of a bit with logic 1, the result is equal to its own value. This procedure must be done to disregard the conditions of other keys of which we do not want to detect the activation. In this

Fig. 6.17 Flowchart and source program in Assembly of the main program regarding Exercise 6.7.1 – II



bit	7	6	5	4	3	2	1	0		Operation
(P0)=	X	X	X	X	X	X	X	X		
constant	0	1	0	0	1	0	0	0	=#48h	AND
result	0	X	0	0	X	0	0	0		

Fig. 6.18 Masking operation (AND logic operation) of port 0 (which presents the electrical conditions of the keys) and the constant 48h (we define this constant by defining logic 1 in the bits corresponding to keys 3 and 6, respectively) that was calculated as a function of the keys of which we want to detect the activations

case, we only have to consider the activations of keys 3 and 6 simultaneously, as the conditions of keys 0, 1, 2, 4, 5, and 7 must not be considered.

Therefore, we have four conditions:

- When both keys 3 and 6 are deactivated (the key interface defines logic 1 to bits 3 and 6)
- When key 3 is activated and key 6 is deactivated (the key interface defines logic 0 to bit 3 and defines logic 1 to bit 6 of port 0)
- When key 3 is deactivated and key 6 is activated (the key interface defines logic 1 to the bit 3 and defines logic 0 to bit 6 of port 0)
- When both keys 3 and 6 are activated (the key interface defines logic 0 to bits 3 and 6)

Figure 6.19 illustrates the different combinations of keys 3 and 6, the result of the AND operation between the content of the P0 register and the constant 48h, and the logic condition of the zero flag.

Based on Fig. 6.19, we can observe that when at least one of the keys is deactivated, the zero flag is equal to logic 0, and when both are activated, the zero flag is equal to logic 1. Therefore, the strategy to detect the activation of both keys in the same time is to analyze the condition of the zero flag, after performing the masking operation (AND logic operation) between the content of port 0 with the constant, which in this case is equal to 48h. Figure 6.20 presents the flowchart and the source program of the subroutine that detects the activations of keys 3 and 6 simultaneously, considering the bounces of the keys.

IV. This software must show a binary down Counter on the LEDs in a blinking mode for a period of 0.6 s, whenever key 2 (which is electrically connected to bit 2 of port 0) is activated (consider bounce).

Solution The only change in the software 6.7.1 – I described above is in the main program, to which we must add the instruction LCALL T600 ms after deactivating the LEDs in the end of the main program, as indicated in Fig. 6.21.

(P0)=	X	1	X	X	1	X	X	X		
constant	0	1	0	0	1	0	0	0	=#48h	AND
result	0	1	0	0	1	0	0	0	=48h	(Z)=0

bit	7	6	5	4	3	2	1	0		Operation
(P0)=	X	1	X	X	0	X	X	X		
constant	0	1	0	0	1	0	0	0	=#48h	AND
result	0	1	0	0	0	0	0	0	=40h	(Z)=0

bit	7	6	5	4	3	2	1	0		Operation
(P0)=	X	0	X	X	1	X	X	X		
constant	0	1	0	0	1	0	0	0	=#48h	AND
result	0	0	0	0	1	0	0	0	=08h	(Z)=0

bit	7	6	5	4	3	2	1	0		Operation
(P0)=	X	0	X	X	0	X	X	X		
constant	0	1	0	0	1	0	0	0	=#48h	AND
result	0	0	0	0	0	0	0	0	=00h	(Z)=1

Fig. 6.19 The different combinations of keys 3 and 6, the result of the AND operation between the content of the P0 register and the constant 48h, and the logic condition of the zero flag (Z)

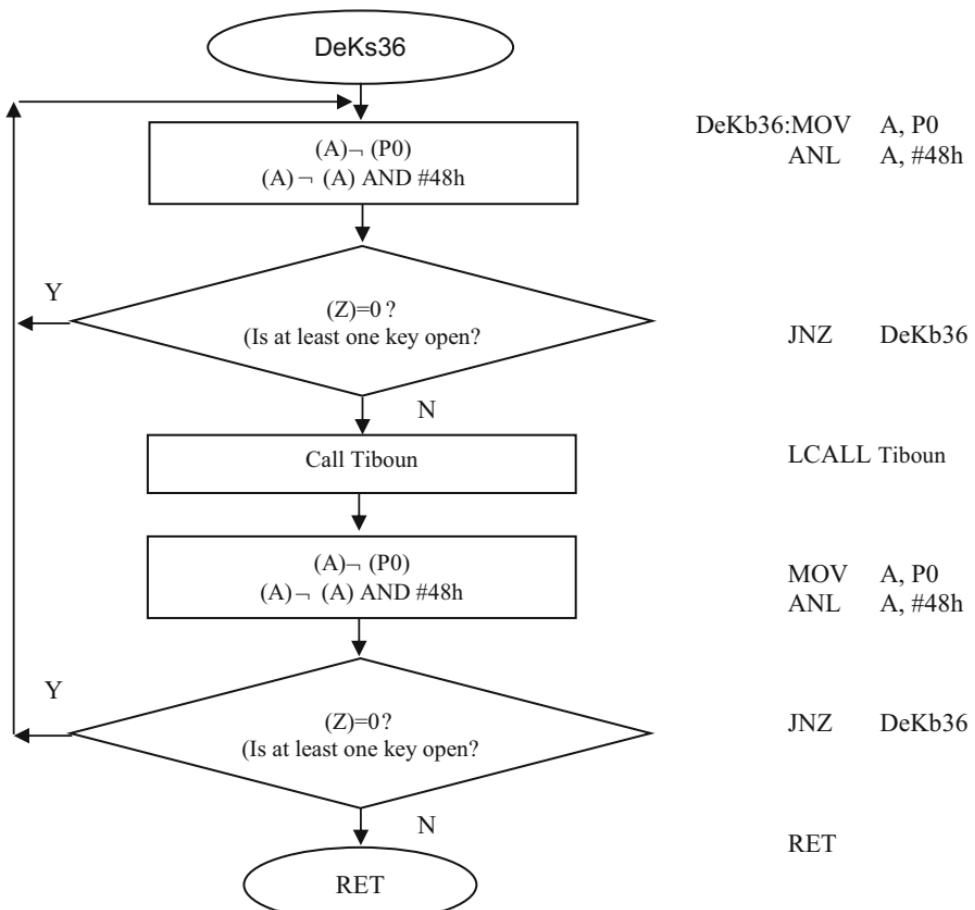


Fig. 6.20 Flowchart and the source program of the subroutine that detect the activations of keys 3 and 6 simultaneously, considering the bounces of the keys

Fig. 6.21 Flowchart and source program in Assembly of the main program regarding Exercise 6.7.1 – IV

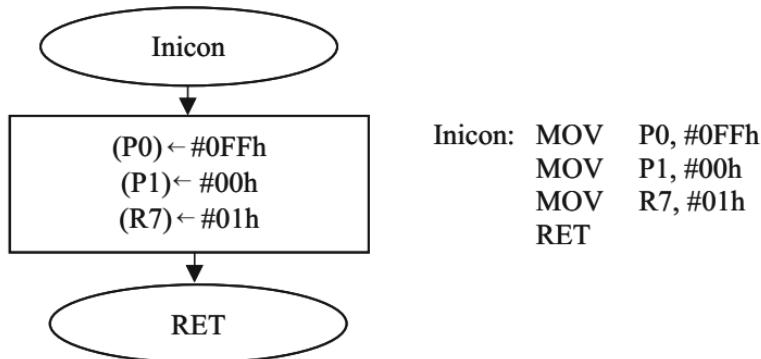
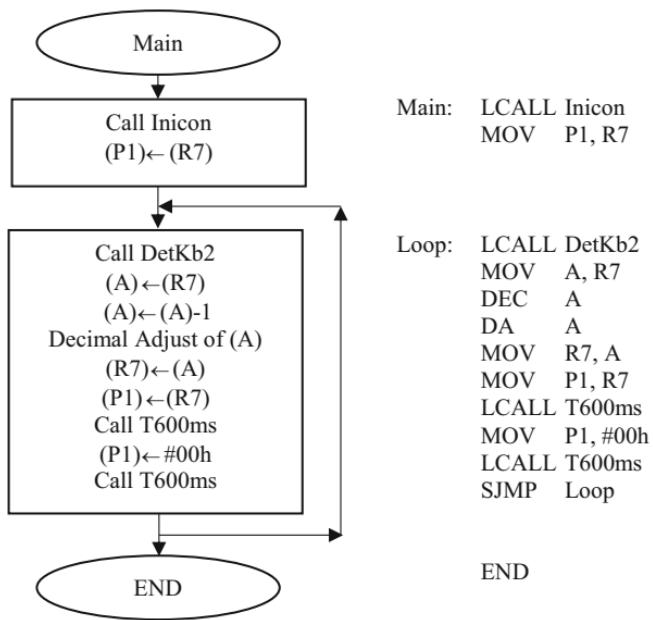
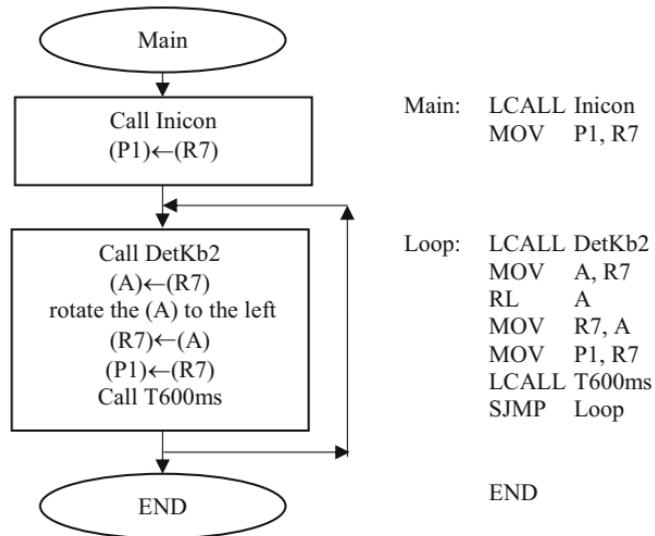


Fig. 6.22 Flowchart and source program that configure the ports and defines the content of the R7 register to be shown on the LEDs (activate bit 0 and deactivate the rest of the bits)

IV. This software must perform a rotation operation of a bit to the left with the content of the port 1 for a period of 0.6 s, whenever the key 2 (which is electrically connected to the bit 2 of port 0) is activated (consider bounce).

Solution The software related to Exercise 6.7.1 – I described above must be changed regarding the subroutine of initialization (Inicon), where the content of the R7 register must be initialized with 01h, and the main program must perform a rotation operation of a bit to the left with the content of the port 1, as indicated in Figs. 6.22 and 6.23.

Fig. 6.23 Flowchart and source program in Assembly of the main program regarding Exercise 6.7.1 – IV



6.8 Proposed Exercises

- 6.8.1. Design the following flowcharts and structured source programs in Assembly using one of the members of the 8051 core microcontroller family, considering that the port 0 is electrically connected to an input interface with a eight-key dipswitch (1, open key; 0, closed key) and that the port 1 is electrically connected to a set of eight LEDs (positive logic: logic 0 deactivates the LED and 1 logic activates the LED), considering the following:
- I. The software must show a binary up Counter on the LEDs for a period of 1 s whenever the key 4 (which is electrically connected to bit 4 of the port 0) is activated (consider bounce).
 - II. The software must show a hexadecimal up Counter of even numbers on the LEDs in a blinking mode for a period of 2 s, whenever the keys 1 and 7 (which are electrically connected to the bits 1 and 7 of the port 0) are activated (consider bounce).
 - III. This software must perform rotation operations of two bits to the right with the content of port 1 for a period of 1.5 s, whenever the keys 0, 2, and 5 (which are electrically connected to the bits 0, 2, and 5 of port 0) are activated (consider bounce).
 - IV. The software must show a decimal up Counter of odd numbers on the LEDs in a blinking mode for a period of 3 s, whenever the keys 2 and 3 (which are electrically connected to the bits 2 and 3 of port 0) are activated (consider bounce).

References

1. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
2. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
3. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
4. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
5. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
6. Atmel Corporation (1997) AT89 Series Hardware Description (0499B-B), Dec 1997
7. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
8. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D-MICRO), June 2008
9. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
10. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

Basic 8051 Core Microcontroller Interruptions

7.1 Introduction

The most common method of communication between humans is by interruptions, e.g., when a person calls/asks another (it is an interruption process). The person who is contacted must usually stop what is doing and must answer to the request of the person that needs to be attended. Afterwards, the person who has been called must return to what was doing. This same approach has also been implemented in computer systems, for instance, when a transducer (electrical/electronic circuit with a sensor) that is electrically connected to one of its interruption inputs requests an action from the computer system. Therefore, the computer system must stop what it is doing, answer the call of the interruption, and finally resume what it was doing. This strategy of input variable monitoring greatly facilitates the use of microcontrollers in machine control applications [1–10].

In this section, we will describe the set of interruptions that exist on the 8051 core microcontrollers from Intel. In addition, the operation process of the interruption system of this microcontroller is described in detail. Some examples of designs that use this interruption system will be provided too [1–10].

7.2 Methods of Variable Management in a Microcontrolled System (Scanning and Interruption)

The 8051 core microcontroller family has at least five interruption sources (six for 8X32/8X52 microcontrollers) [10].

There are two basic techniques to manage an input variable, which is defined by any input interface that is electrically connected to the ports of the microcontroller. The first method is called “by scanning,” in which the programmer develops a routine and places it in the main program loop, and in each cycle (loop) of the main program, the variable must be read and analyzed, enabling it to generate actions that depend on the logic condition of this variable. To illustrate, consider an

application where a cooling system must be activated whenever the temperature exceeds 20 °C. Consequently, the computer system must present two interfaces: one with a temperature transducer (electronic circuit with temperature sensor) and another with an output interface, responsible for activating the cooling system. The output of the circuit of the temperature transducer is electrically connected to one of the bits of the microcontroller ports, which provides a logic 0 value if the temperature is lower than or equal to 20 °C and a logic 1 if the temperature exceeds 20 °C. The programmer must prepare a routine within the main program loop, and at each loop, the program should read the value of that digital variable, which is the temperature in this case. If the value is logic 0, the program does not activate the cooling system; otherwise this routine must activate the cooling system [10].

Another way to manage an input variable is through the interruption method, which, unlike the previous one, is composed of a subroutine to service this interruption source, which must be positioned outside the main program loop, i.e., in a program memory address that is predetermined by the manufacturer. In this case, the external transducer interface of temperature must be connected to one of the external interruption inputs of the microcontroller. Therefore, whenever the temperature transducer generates a logic 1 (the temperature exceeds 20 °C), an interruption is generated to the microcontroller [10].

The check of the occurrence of interruptions of a microcontroller is always performed at each end of the instruction fetch cycle in the program memory. Once the microcontroller recognizes an interruption from an input interface, which can be generated internally (by the timers/counters, serial communication channel, analog/digital converters, digital/analog converters, etc.) or externally (transducers with sensors), it terminates the execution of the instruction (it performs the execution cycle of the instruction) in progress, which may be an instruction of a subroutine or a statement positioned in the main program. If the interruption is enabled to be answered (managed by the special function register entitled “*Interruption Enabler*” – IE), the microprocessor performs its service. This is done automatically via hardware, triggered by the electrical signal of the interruption, while the microprocessor runs an instruction call to the subroutine (*LCALL address defined by the manufacturer*). The service subroutine for this interruption source must be positioned by the programmer at the program memory address that is defined by the manufacturer. Each interruption source requires that its service subroutine be positioned in the program memory address that was defined by the manufacturer. All service subroutine of an interruption source must be terminated with a RETI (Return Interruption) statement. Therefore, when the interruption is detected, the microprocessor finishes the execution of the instruction belonging to the main program and executes the instruction “*LCALL address defined by the manufacturer*”. Then, the processing flow is bypassed to execute the instructions of the service subroutine of the interruption source. At the end, when the RETI instruction of the service subroutine is executed, the microprocessor returns to the main program, processing the instruction immediately after that in which the interruption request was detected. Figure 7.1 illustrates in detail the procedure of attending an interruption by the 8051 core microcontroller [10].

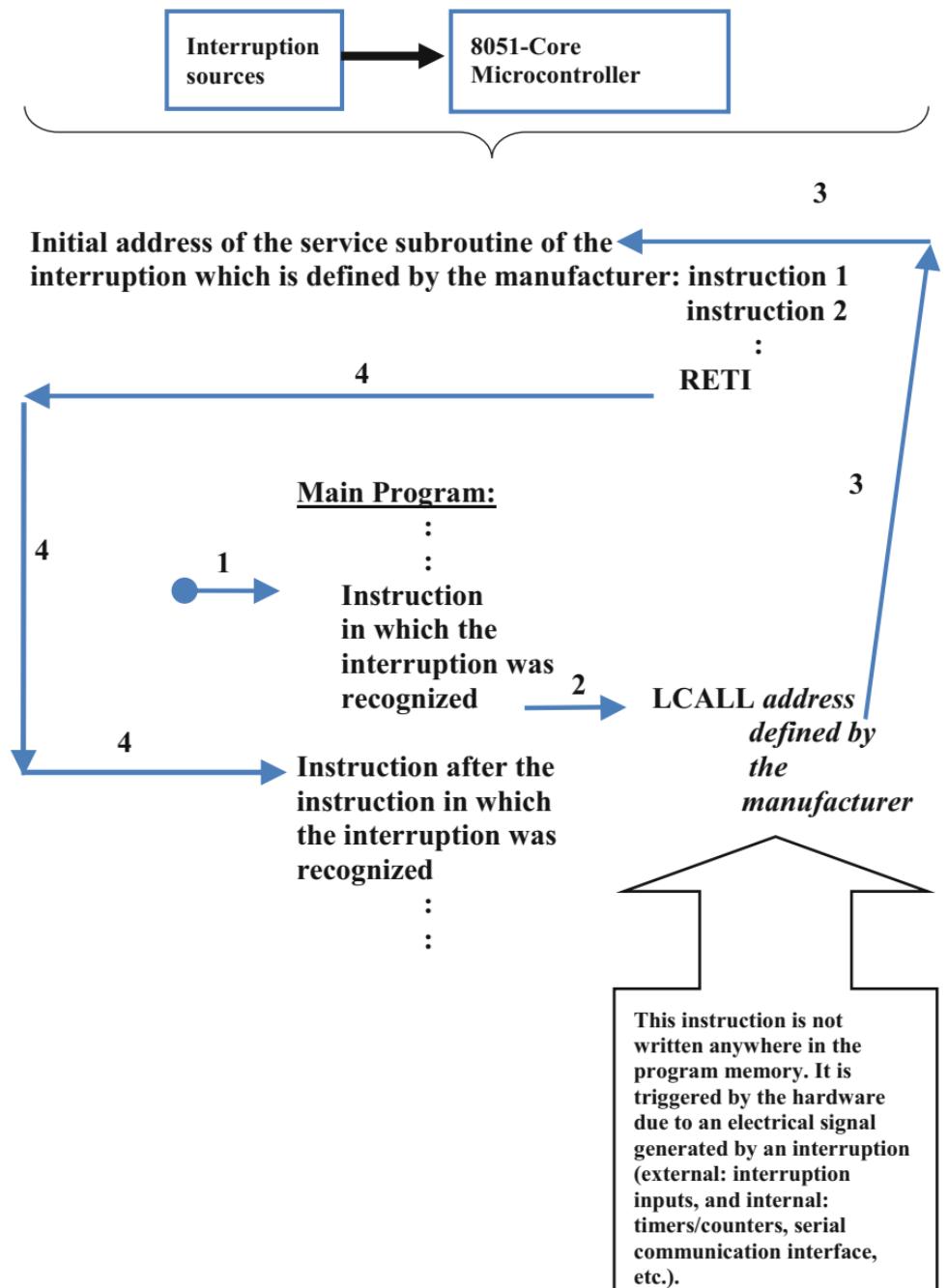


Fig. 7.1 Description of the procedure of attending an interruption by the 8051 core microcontroller

Note that attending to the service subroutine of an interruption source works similarly to the instruction of “calling a subroutine.” The only difference is that the instruction of calling a subroutine by software (**LCALL address**, in which the address is the initial address of the subroutine) must be an integral part of the

Table 7.1 Description of the instruction “LCALL address defined by the manufacturer”

Instruction	Byte	Cycles	Coding	Symbolic representation
LCALL $addr_{16}$ (instruction of the call to the subroutine by software that must be written in the program memory of the main program)	3	2	0001 0010 addr ₁₅₋₈ addr ₇₋₀	(PC) ← (PC) +3 (SP) ← (SP) + 1 ((SP)) ← (PC ₇₋₀) (SP ← (SP) + 1 ((SP)) ← (PC ₁₅₋₈) (PC) ← addr ₁₆
LCALL <i>address defined by the manufacturer</i> (instruction of the call to the service subroutine of an interruption. This instruction is triggered in the hardware by an interruption and it is not written in the program memory)	3	2	0001 0010 addr ₁₅₋₈ addr ₇₋₀	(SP) ← (SP) + 1 ((SP)) ← (PC ₇₋₀) (SP ← (SP) + 1 ((SP)) ← (PC ₁₅₋₈) (PC) ← addr ₁₆

source program, while the instruction of calling to a service subroutine of an interruption (LCALL *address defined by the manufacturer*), which is run when an interruption occurs, is not written anywhere in the program memory because it is automatically executed by the hardware as a consequence of an electrical signal generated by the interruption source. Besides, the instruction “LCALL *address*” defined by the manufacturer does not calculate the address of the next instruction to be executed as “LCALL *address*.” This occurs because the processing flow must return from the service subroutine to the main program regarding the subsequent instruction (one subsequent instruction) where the interruption has been recognized to be attended after running an instruction of the main program. Therefore, the instruction “LCALL *address defined by the manufacturer*” simply stores the return address of the next instruction to be executed on the stack with the return address generated by the last instruction run by the main program. This return address has been generated by the last instruction, which was executed by the main program, when the interruption was recognized and attended. Table 7.1 shows the instructions that are related with the instruction “call to the service subroutine of the interruption source” [1–10].

Therefore, in order to understand the process of attending to an interruption source, it is only necessary to understand the working principle of a call to the subroutine, i.e., to understand how the instructions “LCALL *address*” and “RETI”

work, considering that the RETI instruction presents the same functions of the RET instruction [1–10].

7.3 The Basic Interruption Sources of the 8051 Core Microcontroller

At least five interruption inputs are found on 8051 core microcontrollers from Intel: two external interruptions, two from the two timers/counters, and one from the serial communication interface. Other 8051 core microcontrollers from other manufacturers present additional interruption sources [1–10].

7.3.1 External Interruptions

There are two external interruption inputs in the 8051 core microcontroller, entitled interruption 0 (**INT0**) and interruption 1 (**INT1**). The input pins of the 8051 core microcontroller regarding these external interruptions are electrically connected to the bits 2 and 3 of port 3, i.e., P3.2 and P3.3, respectively. The four least significant bits of the timer control register, called TCON, configure and manage the operation of external interruptions, as described below [1–10].

<i>bits</i>	7	6	5	4	3	2	1	0
	TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0
(TCON)=	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

These are their characteristics:

- **IT0 (TCON.0):** this bit is responsible for configuring how the external interruption 0 (**INT0**) will be recognized by the microcontroller: by level (if this bit is defined with logic 0) or falling edge (if this bit is defined with logic 1).
- **IE0 (TCON.1):** this bit is set ($IE0 = 1$) by the hardware when an external interruption 0 (**INT0**) is detected by P3.2 of the microcontroller. If the external interruption 0 (**INT0**) is configured to generate interruptions by a falling edge, this bit is reset ($IE0 = 0$) when the service subroutine of this interruption source is processed. If the external interruption 0 (**INT0**) is configured to generate interruptions by a low logic level, this bit is not reset when the service subroutine of this interruption source is processed, and therefore it must be reset in the service subroutine of this interruption source.
- **IT1 (TCON.2):** this bit is responsible for configuring how the external interruption 0 (**INT1**) will be recognized by the microcontroller: by level (if this bit is defined with logic 0) or falling edge (if this bit is defined with logic 1).
- **IE1 (TCON.3):** this bit is set ($IE1 = 1$) by the hardware when an external interruption 0 (**INT1**) is detected by P3.2 of the microcontroller. If the external

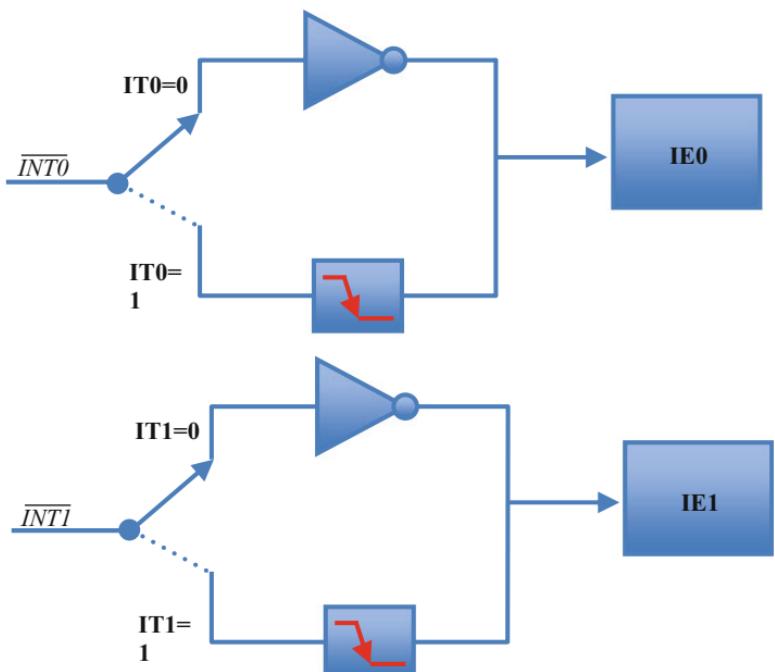


Fig. 7.2 External interruption sources and their control bits in TCON

interruption 0 ($\overline{\text{INT1}}$) is configured to generate interruptions by a falling edge, this bit is reset ($\text{IE1} = 0$) when the service subroutine of this interruption source is processed. If the external interruption 0 ($\overline{\text{INT1}}$) is configured to generate interruptions by a low logic level, this bit is not reset when the service subroutine of this interruption source is processed, and therefore it must be reset in the service subroutine of this interruption source.

Figure 7.2 illustrates the external interruption sources with their respective control bits of the special function register TCON [1–10].

7.3.2 Interruptions of Timers/Counters 0 and 1 of the 8051 Core Microcontroller

A main function of a Timer/Counter is to generate an interruption within a preprogrammed time interval to the 8051 core microcontroller. This interruption is usually used in the time-dependent activity, e.g., to update the time of a clock, to read a data from a keyboard, to write a data in a seven-segment display every 2 ms, to activate/deactivate a LED for each second, etc. [1–10].

The Timer/Counter usually consists of a counting register which is always incremented by one unit when a falling edge of an input clock signal occurs. When the counting register of the Timer/Counter reaches the final counting, and

after another falling edge of the clock signal, it goes back to the initial counting (111...1→000...0), i.e., it overflows and an electrical signal of the interruption occurs, which is stored in the bit TF0 (TCON.5) for the Timer/Counter 0 and (TF1 = TCON.7) for the Timer/Counter 1. TF0 and TF1 of the special function register TCON are responsible for storing the logic conditions of the interruptions 0 and 1, respectively, i.e., they indicate if the interruptions are still not occurred (equal to 0) or if they have already happened (equal to 1). Each time that a Timer/Counter interruption occurs, the attending (service) process to the interruption is the one presented in Fig. 7.1. Consequently, when a Timer/Counter interruption occurs, the microprocessor of the microcontroller finalizes the instruction of the main program that was being run. If the interruption is enabled, the instruction “*LCALL address defined by the manufacturer*” is executed by the hardware, in which the service subroutine is run. When the RETI instruction of the service subroutine is executed, it goes back to run the subsequent instruction of the main program where the interruption was detected [1–10].

The four most significant bits of the special function register called TCON manage the operation of interruptions of the Timers/Counters 0 and 1, as indicated below [1–10].

<i>bits</i>	7	6	5	4	3	2	1	0
	TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0
(TCON)=	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

These are their characteristics:

- **TR0 (TCON.4):** this bit is responsible for stopping/continuing the counting of Timer/Counter 0.
- **TF0 (TCON.5):** it is responsible for storing the logic condition of the interruption of the Timer/Counter 0, i.e., it indicates if the interruption still has not occurred (equal to 0) or if it has already happened (equal to 1). It is set by the hardware whenever Timer/Counter 0 overflows (it goes from the maximum counting to the minimum counting, 11...1→00...0). This flag is reset by the hardware whenever the service subroutine of the interruption source is vectored and executed.
- **TR1 (TCON.6):** this bit is responsible for stopping/continuing the counting of Timer/Counter 1.
- **TF1 (TCON.7):** it is responsible for storing the logic condition of the interruption 0 of the Timer/Counter 1, i.e., it indicates if the interruption still has not occurred (equal to 0) or if it has already happened (equal to 1). It is set by the hardware whenever Timer/Counter 1 overflows (it goes from the maximum counting to the minimum counting, 11...1→00...0). This flag is reset by the hardware whenever the service subroutine of the interruption source is vectored and executed.

Fig. 7.3 Interruption signal of the serial communication interface



7.3.3 The Interruption of the Serial Communication Interface of the 8051 Core Microcontroller

The 8051 core microcontroller presents a serial communication interface, which is responsible for performing the serialization of a byte to be transmitted or to receive serialized bits through the serial communication interface [1–10].

The interruption signal of this interface is generated (logic 1) by an OR logic between the RI and TI flags of the serial communication interface (Fig. 7.3). The RI flag indicates that a serial data was received and mounted in the content of the special function register called SBUF_{in} (byte), and the TI flag indicates that a byte was stored in the content of the special function register called SBUF_{out} (byte) by the programmer, after automatically it is serialized and transmitted by the microcontroller. Neither of these flags are hardware cleaned when the service subroutines for these interruption sources are vectored (addressed) and processed [1–10].

The service subroutine must determine whether the interruption of the serial communication interface was generated by the RI flag (reception) or by the TI flag (transmission). Besides, this service subroutine must also reset (perform a write operation of logic 0) these flags (RI and TI) by software because they are not reset by hardware, when the service subroutine is vectored and processed [1–10].

7.4 Special Function Registers for Handling Interruptions

All bits that generate the different interrupts can be set (write operation with the logic 1) or reset (write operation with the logic 0) by software, producing the same effects than those produced when they are triggered by hardware. This means that the interruptions can be generated by software and those that are still pending can be canceled by software [1–10].

Each of these interruption sources can be individually enabled or disabled by means of some bits from the special function register called IE (0: disables the interruption source, 1: enables the interruption source). There is also a global enable/disable bit called “Enable All,” which is able to enable and disable all interruptions at once [1–10].

When an interruption is enabled, it means that it will be recognized, vectored, and attended by the microprocessor of the microcontroller, i.e., the instruction “LCALL *address defined by the manufacturer*” will be run. Analogously, when an interruption is disabled, it means that it will not be recognized, vectored, and attended by the microprocessor of the microcontroller, i.e., the instruction “LCALL *address defined by the manufacturer*” will not be executed [1–10].

The special function register called Interruption Enable (IE) is responsible for enabling or disabling the interruptions of the 8051 core microcontroller, as it is described below [1–10].

<i>bit</i>	7	6	5	4	3	2	1	0
(IE) =	IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0
	EA	----	ET2	ES	ET1	EX1	ET0	EX0

Symbol	Bit	Function
EA	IE.7	General disabling of all interruptions 0: no interruption is vectored; 1: each interruption source becomes enabled or disabled depending on the logic condition of its corresponding enabler bit (set or reset)
-	IE.6	Reserved
ET2	IE.5	Enabler bit of Timer/Counter 2 that is responsible for enabling/disabling the attending of this interruption (it activates the execution of the instruction “ <i>LCALL address defined by the manufacturer</i> ”)
ES	IE.4	Enabler bit of the serial communication that is responsible for enabling/disabling the attending of this interruption (it activates the execution of the instruction <i>“LCALL address defined by the manufacturer”</i>)
ET1	IE.3	Enabler bit of Timer/Counter 1 that is responsible for enabling/disabling the attending of this interruption (it activates the execution of the instruction <i>“LCALL address defined by the manufacturer”</i>)
EX1	IE.2	Enabler bit of the external interruption 1 that is responsible for enabling/disabling the attending of this interruption (it activates the execution of the instruction <i>“LCALL address defined by the manufacturer”</i>)
ET0	IE.1	Enabler bit of Timer/Counter 0 that is responsible for enabling/disabling the attending of this interruption (it activates the execution of the instruction <i>“LCALL address defined by the manufacturer”</i>)
EX0	IE.0	Enabler bit of the external interruption 0 that is responsible for enabling/disabling the attending of this interruption (it activates the execution of the instruction <i>“LCALL address defined by the manufacturer”</i>)

It is important to highlight that it is not recommended to use reserved bits (IE.6 [1–10]).

Each interruption source can be individually programmed to have one of two (2) priority levels: low and high. This is done through the special function register called IP (Interruption Priority; 0, low priority; 1, high priority), as indicated below [1–10].

<i>bit</i>	7	6	5	4	3	2	1	0
IP=	IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0
	-	-	PT2	PS	PT1	PX1	PT0	PX0

Priority bit = 1 assigns high priority.

Priority bit = 0 assigns low priority.

Symbol	Bit	Function
-	IP.7	Reserved
-	IP.6	Reserved
PT2	IP.5	Priority bit of the interruption source of Timer/Counter 2
PS	IP.4	Priority bit of the interruption source of the serial communication
PT1	IP.3	Priority bit of the interruption source of Timer/Counter 1
PX1	IP.2	Priority bit of the interruption source of the external interruption 1
PT0	IP.1	Priority bit of the interruption source of Timer/Counter 0
PX0	IP.0	Priority bit of the interruption source of the external interruption 0

An interruption source with low priority may be interrupted by a high priority interruption source, whereas an interruption source with high priority cannot be interrupted by an interruption source with low priority. If two interruption sources with different priorities occur at the same time, the one with the highest priority level will be attended. If two interruption sources with the same priority occur at the same time, a service polling sequence determines which interruption source will be attended, according to Table 7.2 [1–10].

The flags relative to each interruption source are sampled during each machine cycle. The sampling to detect interruptions is interpreted (polled) during the next machine cycle, to which the interruption system generates via hardware an instruction “*LCALL address defined by the manufacturer*” to run the appropriate service subroutine to the interruption source [1–10].

Any of the following conditions will block the generation of the instruction “*LCALL address defined by the manufacturer*,” which is hardware generated [1–10]:

- An equal or greater level interruption is already being processed.
- The current interpretation cycle (polling) to detect the interruption source is not in the final cycle of the execution of the instruction being processed. This ensures that the instruction which is in progress will be completed before vectoring any other service subroutine of the interruption source.
- The RETI instruction is being executed or when an instruction is writing to the IE or IP registers.

After an interruption, the instruction “*LCALL address defined by the manufacturer*” is generated by the hardware and places the content of the current program

Table 7.2 The priority level of each interruption source

#	Interruption source	Priority level	
1	IE0	(+++++)	Highest
2	TF0	(++++)	
3	IE1	(+++)	
4	TF1	(++)	
5	RI + TI	(++)	
6	TF2 + EXF2	(+)	Lowest

Table 7.3 Addresses of service subroutines of interruption sources which must be in the program memory, being defined by the manufacturer

Interruption source	Name of the interruption source	Vectorial Address
RESET	Reset	0000h
IE0	Interruption source of the external interruption 0	0003h
TF0	Interruption source of Timer/Counter 0	000Bh
IE1	Interruption source of the external interruption 1	0013h
TF1	Interruption source of Timer/Counter 1	001Bh
RI + TI	Interruption source of the serial communication interface	0023h
TF2 + EXF2	Interruption source of Timer/Counter 2 and external interruption 2	002Bh

counter (PC) register in the stack. Table 7.3 presents the *address defined by the manufacturer* of the instruction LCALL, regarding each interruption source [1–10].

When the interruption source is vectored and serviced, the microprocessor of the microcontroller executes the service subroutine of the interruption source which is stored at the program memory address indicated in Table 7.3 until the RETI instruction is processed [1–10].

The RETI instruction informs the microprocessor that the service subroutine of the interruption source is no longer in progress, so it reads the two bytes at the top of the stack and reloads the content of the program counter (PC) register with the address return to the main program. In this way, the program's execution continues from the instruction where the interruption occurred [1–10].

To use any interruption of the 8051 core microcontroller, the following steps must be followed [1–10]:

- Set to logic 1 the content of the EA bit (Enable All) of the special function register Interruption Enable (IE) in order to allow the enabled interruption sources to be attended.
- Set to logic 1 the contents of the individual enabler bits of each interruption source that we desire to attend.
- The service routines of interruption sources must be placed at the program memory address according to Table 7.3.

In addition, regarding the external interruptions, the pins P3.2 (**INT0**) and P3.3 (**INT1**) must be configured as input, and therefore the output signals of input interfaces (transducers) which are electrically connected to these external interruptions must be designed to operate initially with logic 1. Whenever an event occurs, their outputs must go to the logic level equal to 0, indicating that an interruption occurs [1–10].

7.5 Resolved Exercises

- 7.5.1. Design a piece of software in Assembly that runs a step-by-step program with the use of the external interruption.

Solution It is known that an interruption request is not handled by the microprocessor of the microcontroller while an interruption source of the same priority level is being executed, or while the RETI instruction of a service subroutine has not been executed yet. Therefore, an interruption source cannot be processed until an instruction of the main program is executed. One way to use this step-by-step operation feature is to program an external interruption source, which must be level-enabled. The service subroutine for this interruption source must end with the following encoding:

Code at 0003h

INT0: :

:
JNB P3.2, \$; It waits for (P3.2) to go to logic 1
; (disregarding the bounce of the switch)
JB P3.2, \$; It waits for (P3.2) to go to logic 0
; (disregarding the bounce of the switch)
RETI ; It returns from the service subroutine of the
; source of the interruption to the main
; program

code

So, the pin of the microcontroller corresponding to the bit P3.2 (INT0) must be electrically connected to an interface with a push-button switch, e.g., which it must present a low logic level when it is deactivated and activated, it must present a high logic level. In this way, the CPU will execute the service subroutine for this interruption source and wait until it has a pulsed signal (from logic 0 to logic 1 and logic 0 again). Then it runs the RETI instruction, returns to the main program, executes a single instruction, and immediately a new interruption occurs. Consequently, the CPU executes the service subroutine of the interruption source INT0 again and waits for a new signal pulsed in P3.2 (INT0). This is how the main program instructions are executed one at a time, whenever P3.2 is pressed.

Fig. 7.4 8051 core microcontroller electrically connected to a transducer circuit containing a push-button dipswitch (single-step operation)

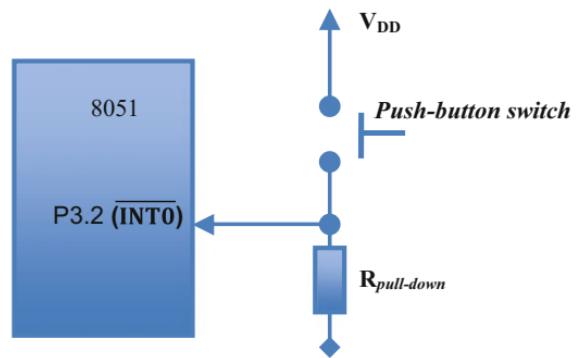


Fig. 7.5 8051 core microcontroller electrically connected to a transducer circuit containing a push-button dipswitch

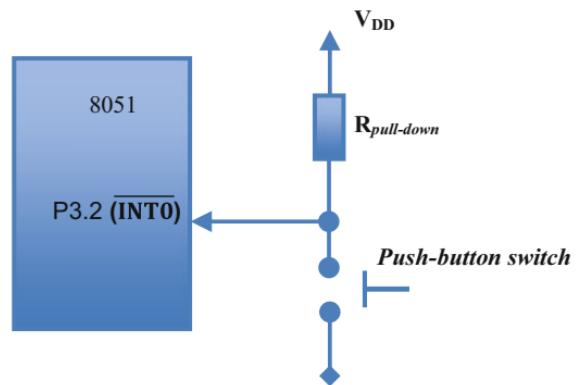


Figure 7.4 shows the block diagram of the circuit to be connected to the external interruption 0 (INT0), which corresponds to the pin P3.2.

7.5.2. Change the previous program so that a LED, which is electrically connected to the bit 5 of P1, is activated when the push-button switch is activated. When another activation of this push-button switch occurs, the LED must be deactivated. This cycle must be maintained whenever the switch is turned on and off.

Solution The output of the transducer circuit with the push-button switch must go to the low logic level when the push-button switch is activated, according to Fig. 7.5.

\$include(REG51.inc)	; It includes the file that contains all special function ; registers of the 8051-core Microcontroller
;	
; After a reset signal code at 0000h	; It defines the instruction ljmp START at the address 0000h ; of the program memory, and then a reset signal the ; microprocessor of the microcontroller jumps to the ; START program address to jump the service subroutine ; of the interruption source (INT0)
code	
;	
; Service subroutine of the external interruption source 0 (INT0). code at 0003h	; 0003h is the location of the program memory where we ; must write the service subroutine of the external ; interruption source (INT0)
CPL P1.5	; When an external interruption 0 (INT0) occurs, i.e. the ; push-button switch is activated, the LED electrically ; connected to P1.5 will change its logic state (in fact, ; the LED flashes as the push-button switch is activated ; twice)
RETI	; It return to the main program, i.e. to the ; AJMP \$ instruction
code	
;	
; Main program code at 0050h	; It sets the main program at the program memory ; address 0050h
START:	MOV P3, #0FFh ; this instruction configures P3 as input MOV P1, #00h ; this instruction configures P1 as output MOV TCON, #00h ; this instruction configures the external interruption ; 0 (INT0) to be activated by a low logic level ; (IT0=0)
MOV IE, #81h	; it enables the external interruption 0 to be attended ; by the microprocessor of the microcontroller
AJMP \$; this instruction jumps to itself (infinite loop). Once ; INT0 happens, LCALL 0003h is executed ; and thus the service subroutine of the external ; interruption source 0 is run. It complements the ; logic level of P1.5, which is electrically ; connected to the led
END	
code	

7.6 Proposed Exercises

- 7.6.1. Design the hardware and software to use the interruption sources 0 (**INT0**) and 1 (**INT1**). The pins corresponding to the external interruption sources 0 (P3.2) and 1 (P3.3) must be electrically connected to the two push-button switches (negative logic). In this hardware, there must be a LED that is electrically connected to bit 0 of port P1. When **INT0** occurs, the LED must be activated, and when **INT1** occurs, the LED must be deactivated.
- 7.6.2. Design the hardware and software to use the interruption sources 0 (**INT0**) and 1 (**INT1**). The pins corresponding to the external interruption sources 0 (P3.2) and 1 (P3.3) must be electrically connected to the two push-button switches (negative logic). In this hardware, there must be 8 LEDs that are electrically connected to port P1. The initial value of the content of P1 must be equal to 01h. When **INT0** occurs, the content of P1 must be rotated one bit to the right, and when **INT1** occurs, the content of P1 must be rotated to the left.
- 7.6.3. Design the hardware and software to use the interruption sources 0 (**INT0**) and 1 (**INT1**). The pins corresponding to the external interruption sources 0 (P3.2) and 1 (P3.3) must be electrically connected to the two push-button switches (negative logic). In this hardware, there must be 8 LEDs that are electrically connected to port P1. The initial value of the content of P1 must be equal to 01h. When **INT0** occurs, the content of P1 must be incremented by one bit, and when **INT1** occurs, the content of P1 must be decremented by one unit.
- 7.6.4. Design the hardware and software to use the interruption sources 0 (**INT0**) and 1 (**INT1**). The pins corresponding to the external interruption sources 0 (P3.2) and 1 (P3.3) must be electrically connected to the two push-button switches (negative logic). In this hardware, there must be 8 LEDs that are electrically connected to port P1. The initial value of the content of P1 must be equal to 01h. When **INT0** occurs, the content of P1 must be incremented by two units, and when **INT1** occurs, the content of P1 must be decremented by two units.

References

1. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
2. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
3. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
4. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
5. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000

6. Atmel Corporation (1997) AT89 Series Hardware Description (0499B–B), Dec 1997
7. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
8. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D-MICRO), June 2008
9. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
10. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

Timers/Counters of the 8051 Core Microcontroller

8.1 Introduction

This chapter is devoted entirely to the Timers/Counters of the 8051 core microcontroller family, which are responsible for generating time and determining the baud rate of the serial communication [1–10].

Several examples are given regarding the use of the Timers/Counters in professional computer systems so that the readers can practice the development of their own projects [1–10].

8.2 Features of the Timers/Counters

Knowing that the process to identify an interruption of Timers/Counters happens at the end of each fetch cycle of an instruction of the main program, and considering that the interruption was enabled previously, when the microprocessor of a microcontroller recognizes its occurrence, it executes an instruction “*LCALL address defined by the manufacturer*,” which is triggered by a hardware signal and is not written anywhere in the software. In the *address defined by the manufacturer* of the program memory, the programmer must write the service subroutine to this interruption. The processing flow changes from main program to the service subroutine of the interruption source, and thus it is attended. When the RETI instruction is executed, the processing flow changes again from the service subroutine of the interruption source to the main program, running one instruction after where the interruption was recognized. Therefore, the process of responding to interruption sources of Timers/Counters is identical to the process of meeting external interruption sources 0 and 1 described in Chap. 7 [1–10].

When a Timer/Counter interruption is generated (overflow of the counting register), an electrical signal is set and stored in a D-type flip-flop (flag), indicating that the interruption happened. This interruption flag is reset by the hardware when its service subroutine is vectored (addressed) [1–10].

8.3 Operation Modes of the Timers/Counters

The Timers/Counters 0 and 1 of the 8051 core microcontroller consist of two counting registers of 16 bits. The low and high significant bytes of the counting registers are called THx and TLx , respectively, where x can assume the values 0 and 1: 0 corresponds to the Timer/Counter 0, and 1 corresponds to the Timer/Counter 1 [1–10].

The two Timers/Counters can be configured to operate either as a Timer or as an Event Counter. When configured as a Timer, the counting register is incremented in every machine cycle, so it can be considered as a machine cycle counter. Since the machine cycle consists of 12 clock periods, the counting ratio is 1/12 of the crystal frequency. When it is set as an Event Counter, the counting register is incremented in response to a transition from an external electrical signal from 1 to 0 (falling edge), which must be connected to the external inputs $\text{T}0$ and $\text{T}1$ of the 8051 core microcontroller. Since it takes two machine cycles (24 clock periods of the external oscillator), the maximum counting ratio is 1/24 of the crystal frequency [1–10].

The clock signal period of the external inputs $\text{T}0$ and $\text{T}1$ must be maintained at least one machine cycle so that they are recognized by the microprocessor [1–10].

The Timers/Counters 0 and 1 can be configured on four different modes of operation: a counting register of 13 bits (Fig. 8.1), a counting register of 16 bits (Fig. 8.2), a counting register of 8 bits with automatic reload (Fig. 8.3), and a

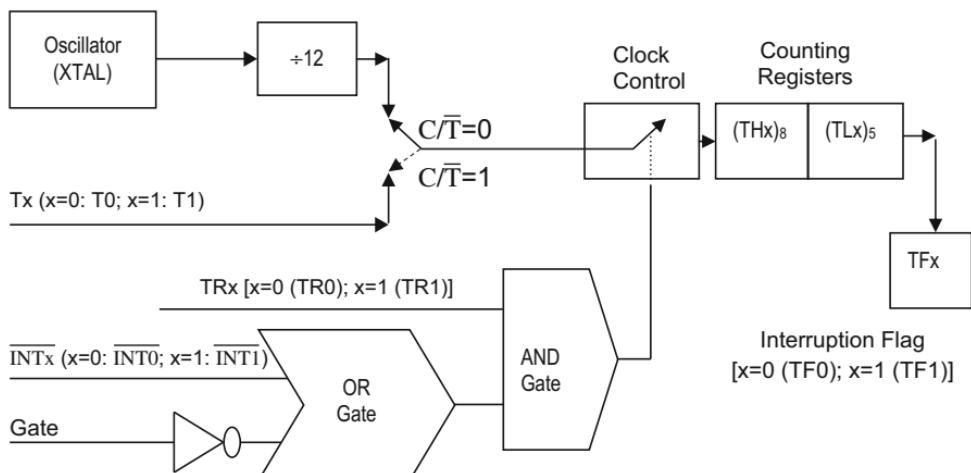


Fig. 8.1 Timer/Counter x , where x can be equal to 0 (Timer/Counter 0) or 1 (Timer/Counter 1) configured on mode 0. It presents a counting register of 13 bits, considering that the 5 least significant bits of the counting must be stored in the $(\text{TLx})_5$, $[0: (\text{TL}0)_5]; 1: [(\text{TL}1)_5]$, and the 8 most significant bits of the counting must be stored in the $(\text{THx})_8$ $[0: [(\text{TH}0)_8]; 1: [(\text{TH}1)_8]$. Bits 5, 6, and 7 of $(\text{TL}0)$ and $(\text{TL}1)$ are not used on this operation mode, and to do a write operation of 8 bits in these registers, we must add three zeros, for example, in the bits 5, 6, and 7 of these registers

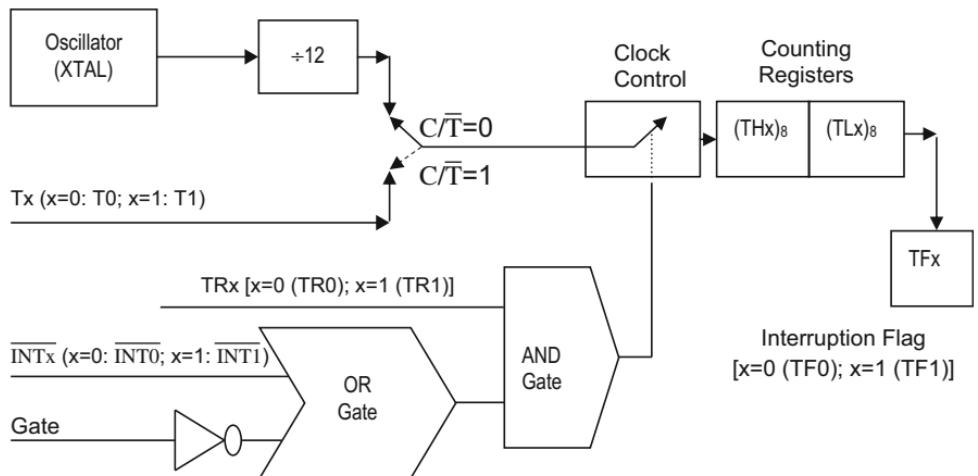


Fig. 8.2 Timer/Counter x, where x can be equal to 0 (Timer/Counter 0) or 1 (Timer/Counter 1) configured on mode 1. It presents a counting register of 16 bits, considering that the 8 least significant bits of the counting must be stored in the $(TLx)_8$, [0: $(TL0)_8$; 1: $(TL1)_8$], and the 8 most significant bits of the counting must be stored in the $(THx)_8$ [0: $(TH0)_8$; 1: $(TH1)_8$]

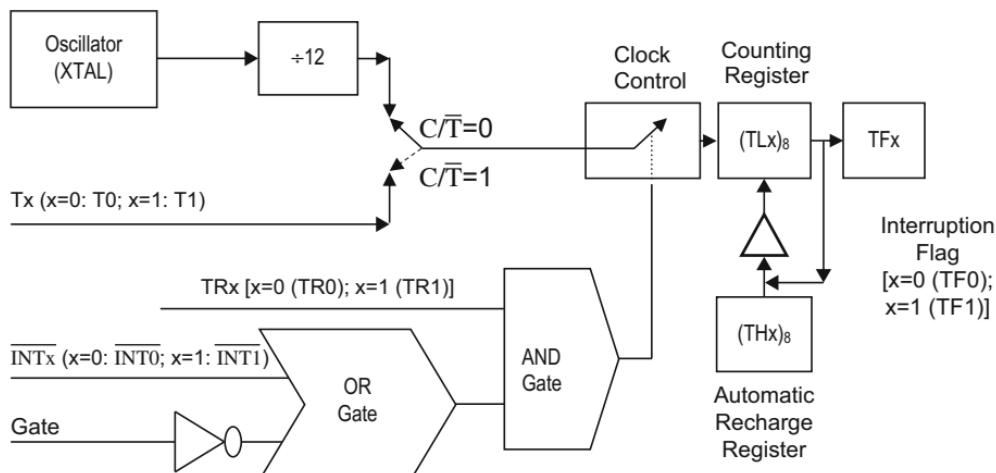


Fig. 8.3 Timer/Counter x, where x can be equal to 0 (Timer/Counter 0) or 1 (Timer/Counter 1) configured in mode 2, which presents a counting register of 8 bits $(TLx)_8$, [0: $(TL0)_8$; 1: $(TL1)_8$] and an automatic recharge register $(THx)_8$ [0: $(TH0)_8$; 1: $(TH1)_8$]

counting register of 8 bits without automatic reload (Fig. 8.4). Besides, Timer Counter 1 can also be configured as a baud rate generator (transmission/reception speed) to the serial communication interface [1–10].

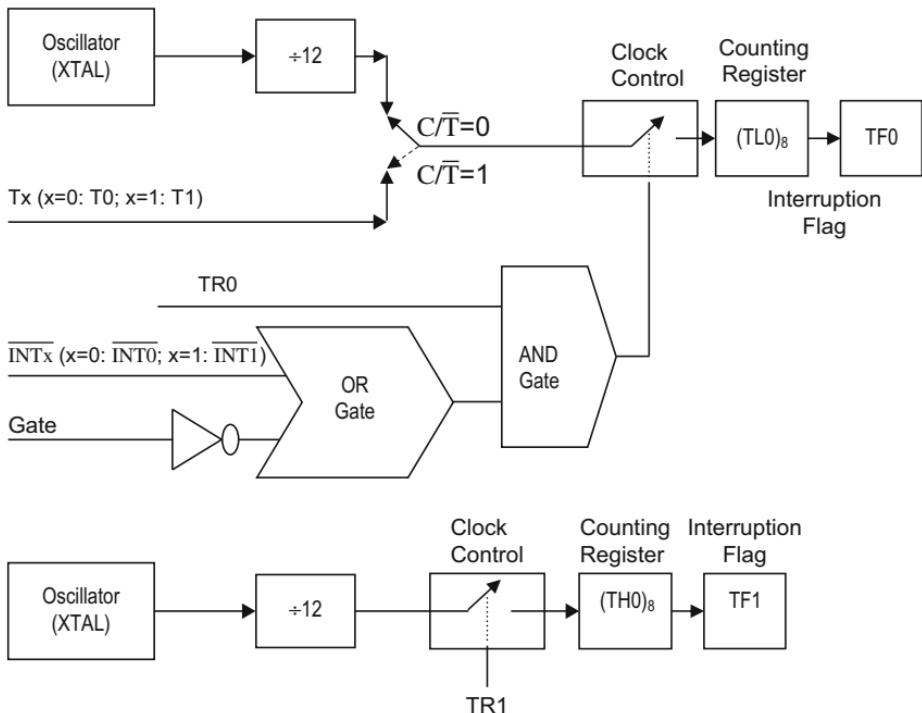


Fig. 8.4 Timer/Counter x, where x can be equal to 0 (Timer/Counter 0) or 1 (Timer/Counter 1) configured on mode 3. Timer/Counter 0 on this operation mode is controlled by its own control bits (C/T , $\overline{INT}0$, Gate, TR0), its counting register is the TL0 (least significant byte of the counting register of Timer/Counter 0), and its interruption bit is the TF0. Timer/Counter 1 on this operation mode is controlled by only a control bit (TR1), its counting register is the TH0 (most significant byte of the counting register of Timer/Counter 0), and its interruption bit is the TF1

8.4 Programming of the Timers/Counters

The special function register named Timer Mode (TMOD) is responsible for programming/configuring the Timers/Counters 0 and 1 [1–10].

	Timer 1					Timer 0			
Bit	7	6	5	4	3	2	1	0	
TMOD =	GATE	C/T	M1	M0	GATE	C/T	M1	M0	

Symbol	Bits	Function
GATE	TMOD.3 TMOD.7	Control bits to enable/disable the counting of the Timers/Counters 0 (TMOD.3) and 1 (TMOD.7) by the hardware: If this bit is equal to a low logic level, the turn-on/turn-off only depends on the TRx, in which x can be equal to 0 or 1 (turn-on/turn-off only by software) If this bit is equal to a high logic level, the turn-on/turn-off depends on the TRx and the logic level of the INTx inputs, in which x can be equal to 0 or 1 (turn-on/turn-off only by software and hardware)
C/T	TMOD.2 TMOD.6	Operation selection of Timers/Counters – Timer or Counter: 0: Select the Timers/Counters to operate as Timer (<i>clock</i> input: f _{crystal} /12) 1: Select the Timers/Counters to operate as Counter (<i>clock</i> input: through the external pin Tx*).
M1	TMOD.4 TMOD.5	They define the operation modes of the Timers/Counters 0 and 1
M0	TMOD.0 TMOD.1	
MI	M0	<i>They program/configure the operation mode of the Timers/Counters 0 and 1</i>
0	0	<i>Mode 0:</i> This mode programs/configures the counting registers to be of 13 bits. Regarding a counting of 13 bits, the initial value must be defined in the 5 least significant bits of TLx (bits 5, 6, and 7 are not used) and the rest of the bits, i.e., the 8 most significant bits of the 13 bits must be defined in the THx. x can be equal to 0 or 1 $\begin{array}{ccccccccc} 1 & 2 & 1 & 1 & 1 & 0 & 9 & 8 & 7 & 6 & 5 \\ \text{Timer with 13 counting bits } (\text{THx}^8=x & x & x & x & xxxx & \text{TLx}^5=***x & xxxx) \\ \text{final counting: } (& 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1) = 8191_{10} \end{array}$ The maximum counting occurs when the initial value is equal to $(\text{THx}) = 0000000_2$ ($\text{TLx} = xxx0000_2$), and therefore the period between interruptions is equal to 8192_{10} multiplied by the period of the input <i>clock</i>
0	1	<i>Mode 1:</i> This mode programs/configures the counting registers to be of 16 bits. Regarding a counting of 16 bits, the initial value must be defined in the 8 least significant bits of the TLx (the 8 bits of this counting register are used) and the rest of the bits, i.e., the 8 most significant bits of the 16 bits must be defined in the THx. x can be equal to 0 or 1 $\begin{array}{ccccccccc} 1 & 5 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ \text{Timer with 16 counting bits } (\text{THx}^8=x & x & x & x & x & x & x & x & \text{TLx}^8=xxxx & xxxx) \\ \text{final counting: } (& 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1) = 65535_{10} \end{array}$ The maximum counting occurs when the initial value is equal to

(continued)

		(THx) = 00000000 ₂ (TLx) = xxx00000 ₂ , and therefore the period between interruptions is equal to 65536 ₁₀ multiplied by the period of the input clock
1	0	<i>Mode 2:</i> This mode programs/configures a counting registers to be of 8 bits (TLx) and another to be an automatic recharge register (THx). The initial value must be defined in the TLx and in the THx. Therefore, the content of the THx is copied to the content of the TLx when it overflows. x can be equal to 0 or 1
1	1	<i>Mode 3:</i> This mode programs/configures the Timers/Counters 0 and 1 to be of 8 bits. The counting registers of the Timers/Counters 0 and 1 are, respectively, the special function registers TL0 and TH0 Timers/Counters 0 is controlled by the control bits of Timer/Counter 0 Timers/Counters 1 is controlled only by the control bit TR1 of Timer/Counter 1

*x can be equal to 0 or 1

The 4 most significant bits of the content of the special function register TCON are responsible for turning on/off and indicating the occurrence of the interruption of the Timers/Counters 0 and 1 [1–10].

Bit	7	6	5	4	3	2	1	0
(TCON) =	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Symbol	Bit	Comment
TF1	TCON.7	This flag indicates the overflow of the Timer/Counter 1. This flag is set by the hardware in the overflow of Timer/Counter 1. This occurs when it goes from the final counting (1...1) to the initial counting (0...0). This bit is reset by the hardware when the interruption is attended, i.e., the microprocessor runs the instruction LCALL 001Bh, where 001Bh is the address in which we must write/record the service subroutine of Timer/Counter 1 in the memory program, which is defined by the manufacturer, in case this interruption is enabled
TR1	TCON.6	This bit is responsible for turning on/off Timer/Counter Count 1 0: Turn-off Timer/Counter Count 1; 1: Turn-on Timer/Counter Count 1. Turn-on Timer/Counter 1 means that it executes the counting operation and Turn-off Timer/Counter 1 means that it stops the counting operation
TF0	TCON.5	This flag indicates the overflow of Timer/Counter 0. This flag is set by the hardware in the overflow of the Timer/Counter 0. This occurs when it goes from the final counting (1...1) to the initial counting (0...0). This bit is reset by the hardware when the interruption is attended, i.e., the microprocessor runs the instruction LCALL 000Bh, where 000Bh is the address in which we must write/record the service subroutine of Timer/Counter 0 in the memory program, which is defined by the manufacturer, in case this interruption is enabled

(continued)

TR0	TCON.4	This bit is responsible for turning on/off Timer/Counter Count 0 0: Turn-off Timer/Counter Count 0; 1: Turn-on Timer/Counter Count 0. Turn-on Timer/Counter 0 means that it executes the counting operation and Turn-off Timer/Counter 1 means that it stops the counting operation
IE1	TCON.3	This bit is set ($IE1 = 1$) by the hardware when an external interruption 0 ($\overline{INT1}$) is detected by P3.2 of the microcontroller. If the external interruption 0 ($\overline{INT1}$) is configured to generate interruptions by a falling edge, this bit is reset ($IE1 = 0$) when the service subroutine of this interruption source is processed. If the external interruption 0 ($\overline{INT1}$) is configured to generate interruptions by a low logic level, this bit is not reset when the service subroutine of this interruption source is processed, and therefore it must be reset in the service subroutine of this interruption source
IT1	TCON.2	This bit is responsible for configuring how the external interruption 0 ($\overline{INT1}$) will be recognized by the microcontroller: by level (if this bit is defined with logic 0) or falling edge (if this bit is defined with logic 1)
IE0	TCON.1	This bit is set ($IE0 = 1$) by the hardware when an external interruption 0 ($\overline{INT0}$) is detected by P3.2 of the microcontroller. If the external interruption 0 ($\overline{INT0}$) is configured to generate interruptions by a falling edge, this bit is reset ($IE0 = 0$) when the service subroutine of this interruption source is processed. If the external interruption 0 ($\overline{INT0}$) is configured to generate interruptions by a low logic level, this bit is not reset when the service subroutine of this interruption source is processed, and therefore it must be reset in the service subroutine of this interruption source
IT0	TCON.0	This bit is responsible for configuring how the external interruption 0 ($\overline{INT0}$) will be recognized by the microcontroller: by level (if this bit is defined with logic 0) or falling edge (if this bit is defined with logic 1)

The control bits of the special function register TMOD (TMOD.2 and TMOD.6) are responsible for setting the Timers/Counters 0 and 1 as a Timer (0) or Counter (1) [1–10].

The Timers/Counters 0 and 1 present four modes of operation (from 0 to 3). These modes are selected by the special function register TMOD (TMOD.0 and TMOD.1 bits for Timer/Counter 0 and TMOD.4 and TMOD.5 for Timer/Counter 1) [1–10].

8.4.1 Mode 0

This operation mode defines that its counting register is of 13 bits. The counting registers are defined by two special function registers: THx and TLx (where x can be 0 or 1), respectively. Both counting registers (THx and TLx) are composed of 8 bits, but only 5 bits (from bit 0 to bit 4) of the TLx are used in this operation mode, i.e., bits 5, 6, and 7 of TLx are not used in this operation mode to store the value of the counting registers. All 8 bits of the THx are used in this operation mode and

therefore totalize a counting register of 13 bits. An initial value of 13 bits must be defined in the counting register to generate a time between the interruptions (T: period). The 8 most significant bits of the initial value must be stored in the THx register, and the 5 least significant bits of the counting must be stored in the 5 least significant bits of the TLx register, as indicated in Fig. 8.1 [1–10].

When the value of the counting of the Timers/Counters overflows, i.e., when it goes from “11111111 xxx11111₂” to “00000000 xxx00000₂,” the interruption flag (TFx) is set (its value is initially equal to logic zero), indicating that an interruption of the Timers/Counters has occurred. Obviously, these interruptions are related to a specific time (time between interruptions), which depends on the input clock and initial values of the counting registers (THx₈ and TLx₅) [1–10].

The maximum time between interruptions of the Timers/Counters 0 and 1 on this operation mode is equal to 2^{13} (=8192₁₀) multiplied by the input clock period ($T_{\text{input_clock}}$), which defines the velocity (frequency: $f_{\text{input_clock}} = 1/T_{\text{input_clock}}$) of the counting of the Timers/Counters. This occurs if the initial value of counting registers is equal to “00000000 xxx00000₂.” To program/configure the Timers/Counters 0 and 1 to generate a time between interruptions (interruptions period) which is smaller than the maximum time between interruptions [8192₁₀. $T_{\text{input_clock}}$], the initial value stored in the contents of the THx₈ and TLx₅ must be equal to 2^{13} subtracted by the desired counting and multiplied by the input clock ($T_{\text{input_clock}}$) of the Timers/Counters 0 and 1. For instance, if the input clock period ($T_{\text{input_clock}}$) of Timer/Counter 0 is equal to 1 μ s and we desire a time between interruptions equal to 5000 μ s (=5 ms), the initial value that we must store in the content of the THx₈ and TLx₅ should be respectively equal to $(2^{13} - 5000)$ ms = (8192₁₀–5000) ms = 3192₁₀, in which we have “01100011 11000₂” when converting it into a binary number of 13 bits. Therefore, in the content of the TL0, we must store the byte xxx11000 (5 least significant bits of the 13 bits). If we consider xxx equal to 000₂ (it could be another binary combination: 001, 010, ..., 111), we have the value 00011000₂ (=18h). In the content of the TH0, we must store the 8 most significant bits of 13 bits found, which is equal to 01100011 = 63h in this case [1–10].

If the interruptions of the Timers/Counters happen and they are enabled regarding the content of the bits ET0 (logic 1) and ET1 (logic 1) of the special function register Interruption Enable (IE), the interruptions are recognized and instructions of “LCALL_address_of_the_service_subroutine_of_Tx_interruption” are generated, where the “address_of_the_service_subroutine_of_Tx_interruption” are equal to 000Bh for the Timer/Counter 0 and 001Bh for the Timer/Counter 1, respectively. These addresses are defined by the manufacturer [1–10].

The Timers/Counters run (perform the addition operation of one unit in the counting registers) only when the TRx (Timer run bits) are equal to logic 1 and GATEx bits are equal to 0. When GATEx bits are equal to the logic 1, the counting of the Timers/Counters x are controlled by the external signals ($\overline{\text{INTx}}$). To illustrate, one of the applications of the Timers/Counters with this configuration are the measurements of the square electric signals width (pulses) generated by barcode readers (transductors composed by a photo-emitter and a photo-receiver) in order to transform white/black narrow and long bars into counting. Afterwards, this counting

is transformed into bits and then into numeric values related to the code bar. Vide the magnetic code interfaces (transductors composed by a magnetic reader), distance measures, velocity measures, etc. [1–10].

8.4.2 Mode 1

When configured on mode 1 (Fig. 8.2), the Timers/Counters 0 and 1 present the same hardware illustrated in Fig. 8.1. The only difference is that the counting registers are of 16 bits. The initial value of 16 bits must be stored in the 8 bits of the contents of the THx and TLx registers. These operation modes are commonly used to generate times between interruptions which are greater than those described on mode 0. The maximum counting allowed for this operation mode is of 2^{16} ($=65536_{10}$), and consequently the maximum time between interruptions that these Timers/Counters are able to generate is 65536_{10} multiplied by $T_{\text{input_clock}}$. The same approach described to mode 0, regarding the programming of the Timers/Counters, must be applied to this operation mode, except that in this case we must transform the initial values found in the counting registers into 2 bytes, instead of 13 bits [1–10].

8.4.3 Mode 2

On this operation mode, the counting registers of the Timers/Counters 0 and 1 are configured to operate with 8 bits. The special function registers (TLx) are the counting registers. The special function registers THx in this operation mode are defined to operate as “automatic recharge registers.” Consequently, when interruptions occur, the content of these “automatic recharge registers” are copied to the counting registers (TLx) with their respective initial values by hardware automatically, as illustrated in Fig. 8.3. Therefore, on this operation mode, the counting registers of these Timers/Counters do not need to be reinitialized in the service subroutines of these interruptions, i.e., this reinitialization process of the counting registers is performed by the hardware automatically when the interruptions occur [1–10].

The maximum counting allowed for this operation mode is of 2^8 ($=256_{10}$), and consequently the maximum time between interruptions that these Timers/Counters are able to generate is 256_{10} multiplied by $T_{\text{input_clock}}$. The same approach described to the modes 0 and 1, regarding the programming of the Timers/Counters, must be applied to this operation mode, except that in this case we must transform the initial values found of the counting registers into 1 byte, instead of 13 and 16 bits [1–10].

When the automatic recharging process occurs, the content of the THx does not change [1–10].

8.4.4 Mode 3

On this operation mode, two independent 8-bit Timers/Counters are generated. The counting registers of the Timers/Counters 0 and 1 are the special function registers TL0 and TH0, respectively. Timer/Counter 0 practically presents the same hardware architecture as those presented on the modes 0, 1, and 2, according to Fig. 8.4. However, Timer/Counter 1 is controlled only by the bit TR1, and it only presents one single option for $T_{\text{input_clock}}$, which is equal to the crystal frequency divided by 12 in this case [1–10].

On this operation mode, Timer/Counter 1 cannot be used as a baud rate generator for the serial communication interface [1–10].

Tables 8.1, 8.2, 8.3, and 8.4 provide some programming/configuration values for the special function register (TMOD) that can be used to program/configure the Timers/Counters 0 and 1 on different operating modes [1–10].

Table 8.1 Timer/Counter 0 operating as a timer

		TMOD	
Mode	Operation mode of Timer/Counter 0	Internal control (Observation 1)	External control (Observation 2)
0	Timer/Counter of 13 bits	00h	08h
1	Timer/Counter of 16 bits	01h	09h
2	Timer/Counter of 8 bits with automatic recharge	02h	0Ah
3	2 Timers/Counters of 8 bits	03h	0Bh

Table 8.2 Timer/Counter 0 operating as a counter

		TMOD	
Mode	Operation mode of Timer/Counter 0	Internal control (Observation 1)	External control (Observation 2)
0	Timer/Counter of 13 bits	04h	0Ch
1	Timer/Counter of 16 bits	05h	0Dh
2	Timer/Counter of 8 bits with automatic recharge	06h	0Eh
3	2 Timers/Counters of 8 bits	07h	0Fh

Table 8.3 Timer/Counter 1 operating as a timer

		TMOD	
Mode	Operation mode of Timer/Counter 1	Internal control (Observation 1)	External control (Observation 2)
0	Timer/Counter of 13 bits	00h	80h
1	Timer/Counter of 16 bits	10h	90h
2	Timer/Counter of 8 bits with automatic recharge	20h	A0h
3	2 Timers/Counters of 8 bits	30h	B0h

Table 8.4 The Timer/Counter 1 operating as a counter

Mode	Operation mode of Timer/Counter 1	TMOD	
		Internal control (Observation 1)	External control (Observation 2)
0	Timer/Counter of 13 bits	40h	C0h
1	Timer/Counter of 16 bits	50h	D0h
2	Timer/Counter of 8 bits with automatic recharge	60h	E0h
3	2 Timers/Counters of 8 bits	70h	F0h

Observation 1

The Timers/Counters are turned on/turned off by setting/resetting the TRx bits ($x = 0$ ou 1) by software.

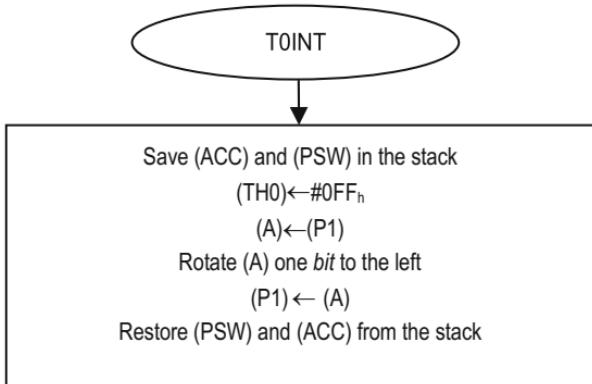
Observation 2

The Timers/Counters are turned on/turned off by the transition from logic 1 to logic 0 (falling edge) in the $\overline{\text{INT}x}$ (P3.2) pin when $\text{TR}x$ ($x = 0$ ou 1) is equal to logic 1 [control of turning on/turning off by the hardware ($\overline{\text{INT}x}$) and software ($\text{TR}x$)].

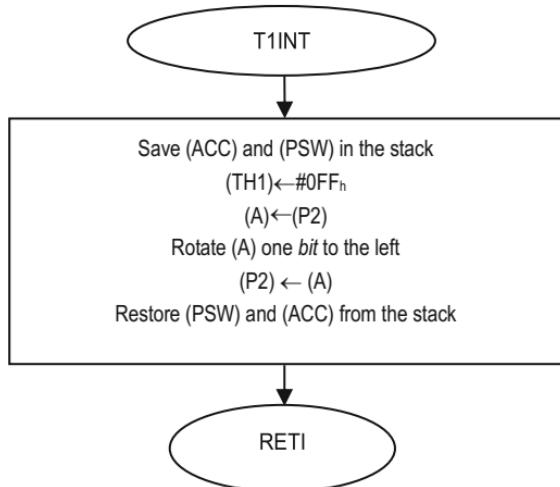
8.5 Solved Exercises

1. To implement a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core-based microcontroller family that is able to perform the following activities:
 - (a) Set the Timer/Counter 0 to the mode 0.
 - (b) Set the Timer/Counter 1 to the mode 1.
 - (c) Both counting registers of the Timers/Counters must be initialized with the value FFh in the contents of the TH0 and TH1 registers and the contents of the TL0 and TL1 registers must be equal to zero.
 - (d) Implement a subroutine which defines the port 0 (P0) as an ascending binary counter. Besides, the value of the content of the port 3 (P3) must be equal to the complement of one of the content of the P0. This subroutine must be stored in the program memory address 0080h.
 - (e) The service subroutine of the Timer/Counter 0 interruption source must rotate the content of the port 1 (P1) 1 bit to the left. This service subroutine must be stored in the program memory address 0040h.
 - (f) The service subroutine of the Timer/Counter 1 interruption source must rotate the content of the port 2 (P2) 1 bit to the left. This service subroutine must be stored in the program memory address 0060h.
 - (g) The main program must be stored in the program memory address 0100h.

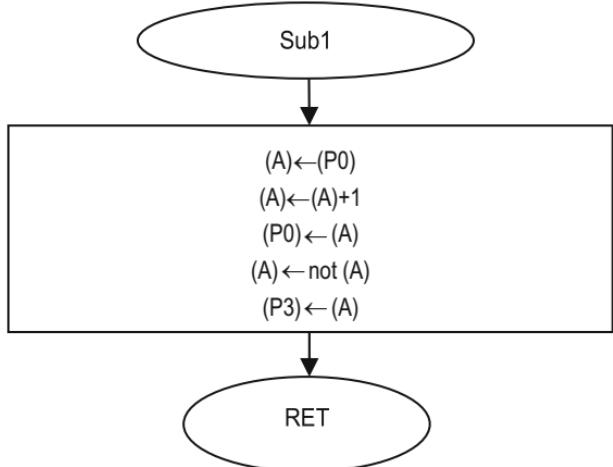
Solution



```
T0INT: PUSH ACC  
        PUSH PSW  
        MOV TH0, #0FFh  
        MOV A, P1  
        RL A  
        MOV P1, A  
        POP PSW  
        POP ACC  
        RETI
```

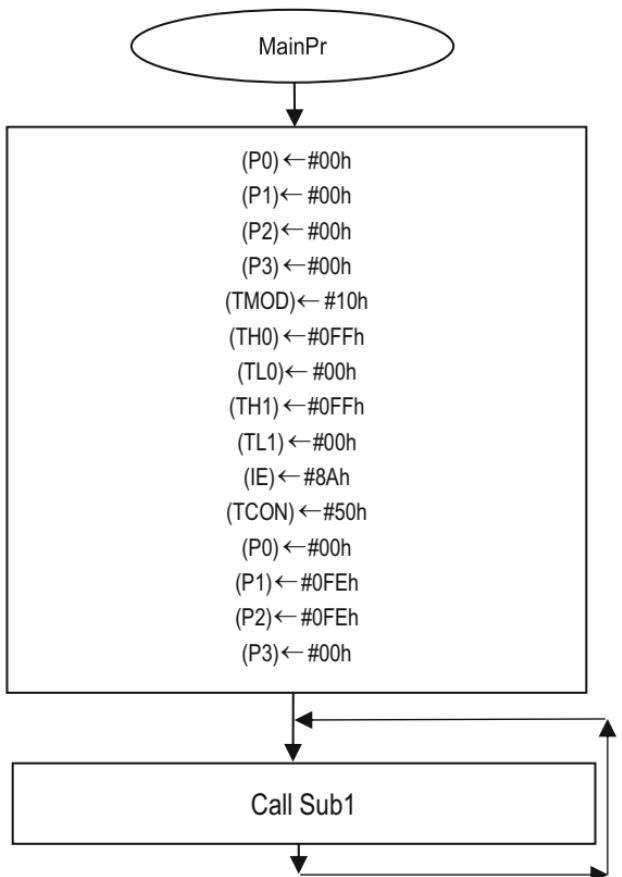


```
T1INT: PUSH ACC  
        PUSH PSW  
        MOV TH1, #0FFh  
        MOV A, P2  
        RL A  
        MOV P2, A  
        POP PSW  
        POP ACC  
        RETI
```



```

SUB1: MOV A, P0
      INC A
      MOV P0, A
      CPL A
      MOV P3, A
      RET
  
```



```

MainPr: MOV P0, #00h
        MOV P1, #00h
        MOV P2, #00h
        MOV P3, #00h
        MOV TMOD, #10h
        MOV TH0, #0FFh
        MOV TL0, #00h
        MOV TH1, #0FFh
        MOV TL1, #00h
        MOV IE, #8Ah
        MOV TCON, #50h
        MOV P0, #00h
        MOV P1, #0FEh
        MOV P2, #0FEh
        MOV P3, #00h
  
```

Loop:

```

    LCALL Sub1
    SJMP Loop
  
```

END

Address of the memory program	Mnemonic	Argument	Comments
ORG	0000h		; It uses the Assembly guideline: “ORG ; 0000h” or “Code at ; 0000h” to write the instruction LJMP ; 0100h in the address ; 0000h. It must be written in the first column: ; ORG 0000h ; LJMP 0100h ; Or ; Code at 0000h ; LJMP 0100h ; Code
0000h	LJMP	0100h	; It jumps to the MainPr, whose address is in 0100h
ORG	000Bh		; It uses the Assembly guideline: “ORG ; 000Bh” or “Code at ; 000Bh … Code” to write the instruction LJMP 0040h in ; the address 000Bh. ; It must be written in the first column: ; ORG 000Bh ; LJMP 0040h ; Or ; Code at 000Bh ; LJMP 0040h ; Code
000Bh	LJMP	0040h	; It jumps to the T0INT, whose address is in 0040h
ORG	001Bh		; It uses the Assembly guideline: “ORG ; 001Bh” or “Code at ; 001Bh … Code” to write the instruction LJMP 0060h in ; the address 001Bh. ; It must be written in the first column: ; ORG 001Bh ; LJMP 0060h ; Or ; Code at 001Bh ; LJMP 0060h ; Code
001Bh	LJMP	0060h	; It jumps to the T0INT, whose address is in 0060h
ORG	0040h		; It uses the Assembly guideline: “ORG ; 0040h” or “Code at ; 0040h … CODE,” as described above, to ; write the ; instructions of the service subroutine of ; the ; Timer/Counter 0 interruption (T0INT) in ; 0040h.

(continued)

T0INT:	PUSH	ACC	; As (ACC) will also be used by this service ; subroutine, ; we must save it in the stack in order to ; preserve its value ; that was defined by the main program.
	PUSH	PSW	; As (PSW) will be changed by this service ; subroutine ; due to the instruction RL A, we must save ; it in the stack ; in order to preserve its value that was ; defined by the main ; program.
	MOV	TH0, #0FFh	; After the overflow of Timer/Counter ; 0, the (TH0) and ; (TL0) are reset (become equal to zero). ; Therefore, we ; must reinitialize them with their initial ; values so that ; the interruption generates the same time ; between ; interruptions again, otherwise the time ; between ; interruptions of Timer/Counter 0 changes. ; In this ; case, (TL0) is reinitialized because its ; initial value is ; equal to zero and thus this procedure is not ; necessary for (TL0), only for (TH0).
	MOV	A, P1	; It reads (P1) to the content of the ; Accumulator (A or ; ACC)
	RL	A	; It rotates (A)
	MOV	P1, A	; It copies (A) to (P1)
	POP	PSW	; It recovers (PSW) from the stack (value ; defined by the ; main program) because it was changed in ; this service ; subroutine. This must be avoided because ; this would ; cause a malfunction in the control that is ; being ; performed in the main program.
	POP	ACC	; It recovers (ACC) from the stack (value ; defined by the ; main program) because it was changed in ; this service ; subroutine. This must be avoided because ; this would ; cause a malfunction in the control that is ; being ; performed in the main program.

(continued)

	RETI		; It returns to the main program
ORG	0060h		; It uses the Assembly guideline: "ORG ; 0060h" or "Code at ; 0060h ... CODE," as described above, to ; write the ; instructions of the service subroutine of ; the ; Timer/Counter 1 interruption (T1INT) in ; 0060h.
T1INT:	PUSH	ACC	; As (ACC) will also be used by this service ; subroutine, ; we must save it in the stack in order to ; preserve its value ; that was defined by the main program.
	PUSH	PSW	; As (PSW) will be changed by this service ; subroutine ; due to the instruction RL A, we must save ; it in the stack ; in order to preserve its value that was ; defined by the main ; program.
WAIT1:	MOV	TH1, #0FFh	; After the overflow of Timer/Counter ; 0, the (TH1) and ; (TL1) are reset (become equal to zero). ; Therefore, we ; must reinitialize them with their initial ; values so that ; the interruption generates the same time ; between ; interruptions again, otherwise the time ; between ; interruptions of Timer/Counter 0 changes. ; In this ; case, (TL1) is reinitialized because its ; initial value is ; equal to zero and thus this procedure is not ; necessary for (TL1), only for (TH1).
	MOV	A, P2	; It reads (P2) to the content of the ; Accumulator (A or ; ACC)
	RL	A	; It rotates (A)
	MOV	P2, A	; It copies (A) to (P2)
	POP	PSW	; It recovers (PSW) from the stack (value ; defined by the ; main program) because it was changed in ; this service ; subroutine. This must be avoided because ; this would ; cause a malfunction in the control that is ; being ; performed in the main program.
	POP	ACC	; It recovers (ACC) from the stack (value ; defined by the

(continued)

			; main program) because it was changed in ; this service ; subroutine. This must be avoided because ; this would ; cause a malfunction in the control that is ; being ; performed in the main program.
	RETI		; It returns to the main program
ORG	0080h		; It uses the Assembly guideline: “ORG ; 0080h” or “Code at ; 0080h . . . CODE,” as described above, to ; write the ; instructions of the subroutine Sub1 in ; 0080h.
Sub1:	MOV	A, P0	; It copies (P0) to (A)
	INC	A	; It increments (A)
	MOV	P0, A	; It copies (A) to (P0)
	CPL	A	; It complements (A)
	MOV	P3, A	; It copies (A) to (P3)
	RET		; It returns to the main program
ORG	0100h		; It uses the Assembly guideline: “ORG ; 0100h” or “Code at ; 0100h . . . CODE,” as described above, to ; write the ; instructions of the subroutine Sub1 in ; 0100h.
MainPr:	MOV	P0, #00h	; It programs/configures port 0 as output
	MOV	P1, #00h	; It programs/configures port 1 as output
	MOV	P2, #00h	; It programs/configures port 2 as output
	MOV	P3, #00h	; It programs/configures port 3 as output
	MOV	TMOD, #10h	; It programs/configures the Timers ; Counters 0 and 1 on the ; modes 0 ($M1 M0 = 00_2$) and ; 1 ($M1 M0 = 01_2$), respectively, ; T_{input_clock} of $1\mu s = 12/f_{crystal} [(C/\bar{T})$; = 0], and turn-on/off by ; software (Gate = 0), as required.
	MOV	TH0, #0FFh	; It initializes the counting register of ; Timer/Counter 0
	MOV	TL0, #00h	
	MOV	TH1, #0FFh	; It initializes the counting register of ; Timer/Counter 1
	MOV	TL1, #00h	
	MOV	IE, #8Ah	; It enables the attending of Timers/ ; Counters 0 and 1
	MOV	TCON, #50h	; It turns on the Timers/Counters 0 and 1

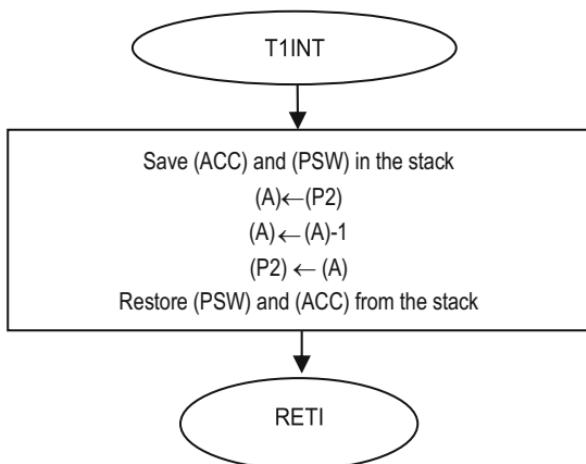
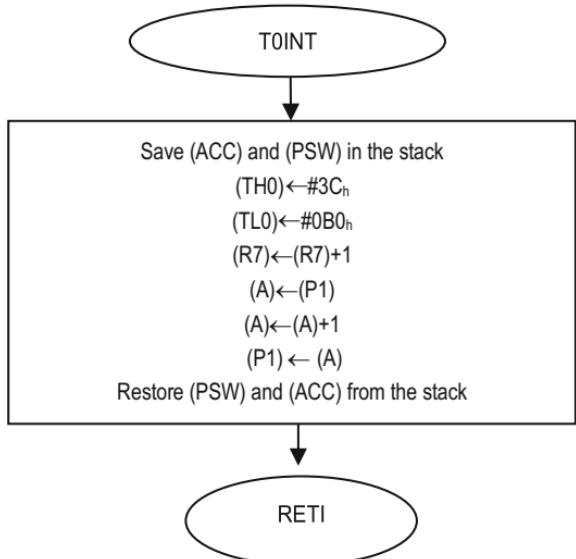
(continued)

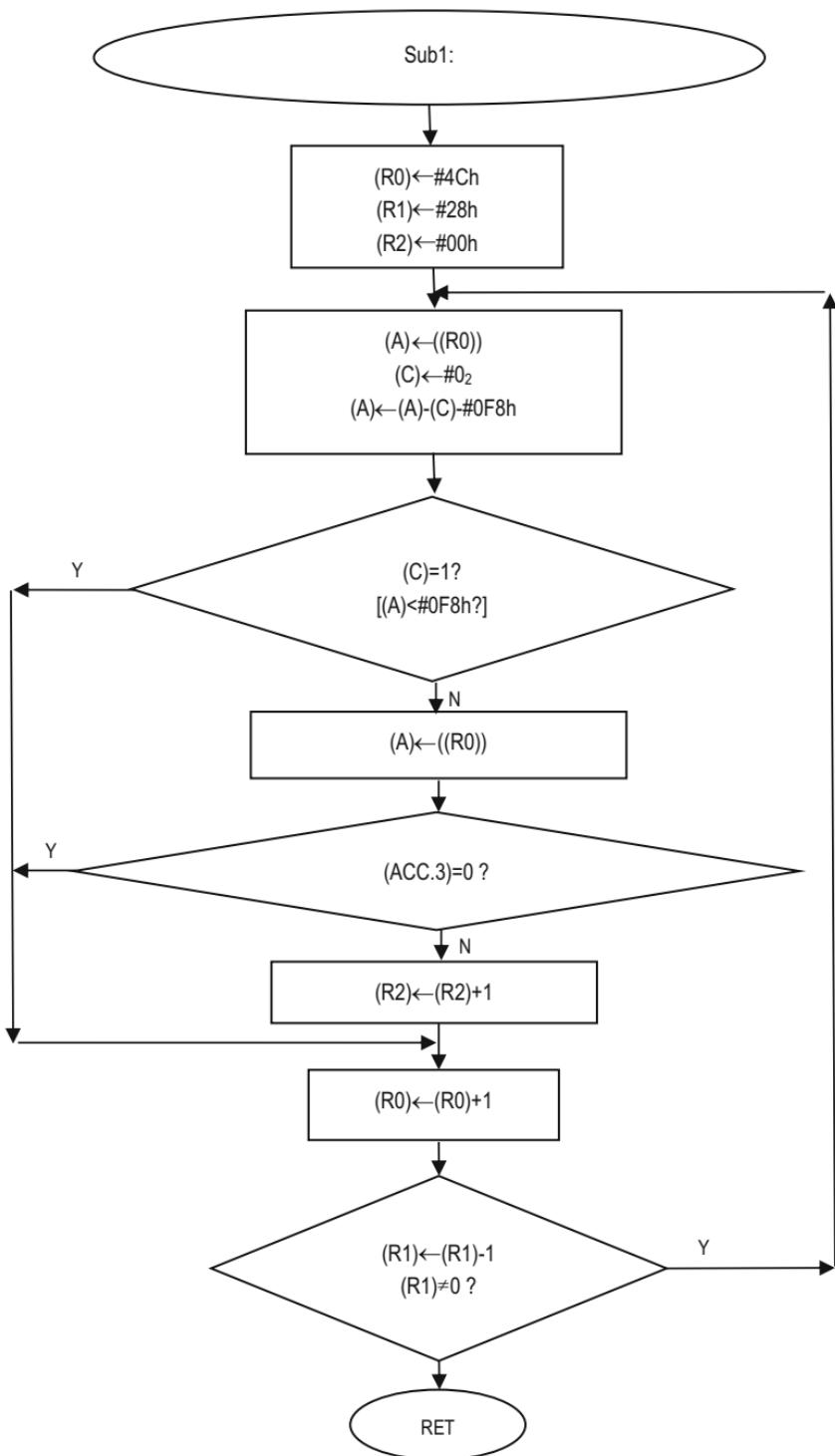
	MOV	P0, #00h	; It resets (P0)
	MOV	P1, #01h	; It initializes (P1) to perform the rotation ; operation ; (only 1 bit is set so that we can see the ; activations of ; the LEDs being rotated to the left).
	MOV	P2, #01h	; It initializes (P2) to perform the rotation ; operation ; (only 1 bit is set)
	MOV	P3, #00h	; It resets (P0)
Loop:	LCALL	Sub1	; It calls Sub1
	SJMP	Loop	; It jumps to the program memory address ; Loop
	END		

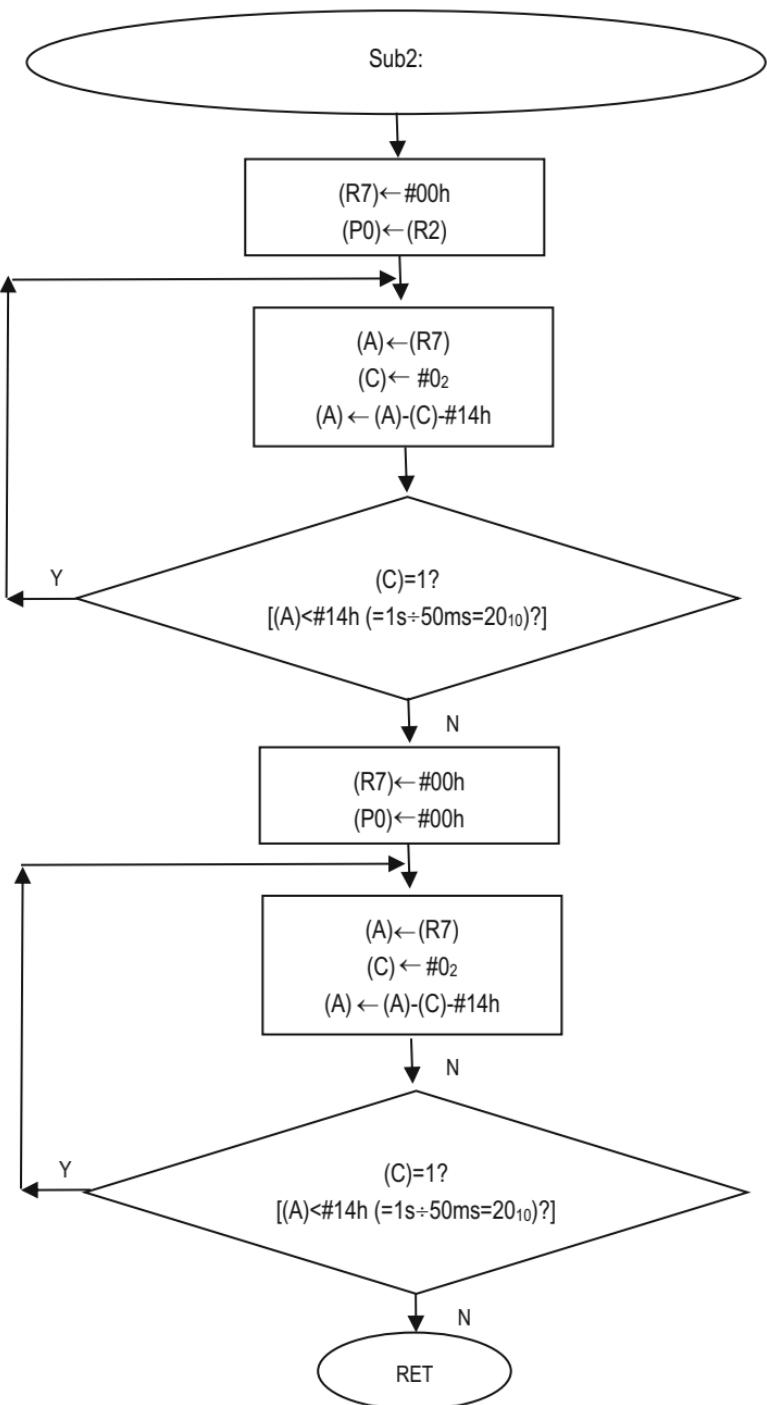
2. Design a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core microcontroller family which is able to perform the following activities:

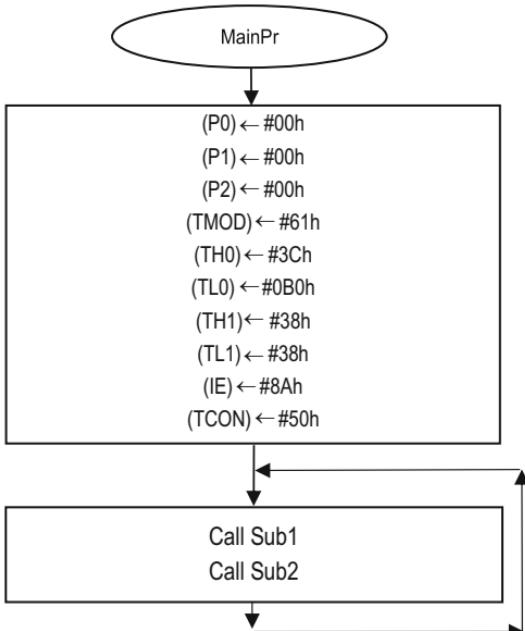
- (a) Program configure the Timer/Counter 0 to the mode 1. It must generate interruptions every 50 ms.
- (b) Program configure the Timer/Counter 1 to the mode 2. It must generate interruptions every 200 ms.
- (c) Implement a subroutine that is able to compute the quantity of elements which are higher than F7h and present the bit 3 equal to 1 of the memory buffer whose initial and final addresses are 4Ch to 73h, respectively. The result must be put in the content of the port 0 (P0) for 1 s (use Timer/Counter 0 to generate the time of 1 s). This subroutine must be stored in the program memory address 0090h.
- (d) Implement a service subroutine in the interruption source of Timer/Counter 0 that transform the port 1 (P1) into an ascendant counter of hexadecimal numbers. This subroutine of service must be stored in the program memory address 0050h.
- (e) Implement a service subroutine in the interruption source of Timer/Counter 1 that transform the port 2 (P2) into a descendant counter of binary numbers. This subroutine of service must be stored in the program memory address 0070h.
- (f) The main program must be stored in the program memory address 0200h.

Solution









MainPr:	MOV	P0, #00h
	MOV	P1, #00h
	MOV	P2, #00h
	MOV	TMOD, #61h
	MOV	TH0, #3Ch
	MOV	TL0, #0B0h
	MOV	TH1, #38h
	MOV	TL1, #38h
	MOV	IE, #8Ah
	MOV	TCON, #50h
Loop:	LCALL	Sub1
	LCALL	Sub2
	SJMP	Loop
		END

Address of the memory program	Mnemonic	Argument	Comments
ORG	0000h		; It uses the Assembly guideline: "ORG 0000h" ; ; or ; "Code at 0000h" to write the instruction LJMP ; 0200h in the address 0000h. It must be written ; in the first column: ; ORG 0000h ; LJMP 0200h ; Or ; Code at 0000h ; LJMP 0200h ; Code
0000h	LJMP	0200h	; It jumps to MainPr, whose address is in ; 0200h
ORG	000Bh		; It uses the Assembly guideline: "ORG 000Bh" ; ; or ; "Code at 000Bh ... Code" to write the ; instruction ; LJMP 0050h in the address 000Bh ; It must be written in the first column: ; ORG 000Bh ; LJMP 0050h ; Or ; Code at 000Bh ; LJMP 0050h ; Code

(continued)

000Bh	LJMP	0050h	; It jumps to T0INT, whose address is in 0050h
ORG	001Bh		<p>; It uses the Assembly guideline: “ORG 001Bh” ; or ; “Code at 001Bh . . . Code” to write the ; instruction ; LJMP 0070h in the address 001Bh. ; It must be written in the first column: ; ORG 001Bh ; LJMP 0070h ; Or ; Code at 001Bh ; LJMP 0070h ; Code</p>
001Bh	LJMP	0070h	; It jumps to T0INT, whose address is in 0070h
ORG	0050h		<p>; It uses the Assembly guideline: “ORG 0050h” ; or ; “Code at 0050h . . . CODE,” as described ; above, ; to write the instructions of the service ; subroutine ; of the Timer/Counter 0 interruption (T0INT) in ; 0050h.</p>
T0INT:	PUSH	ACC	<p>; As (ACC) will also be used by this service ; subroutine, we must save it in the stack in order ; to preserve its value that was defined by the ; main program</p>
	PUSH	PSW	<p>; As (PSW) will be changed by this service ; subroutine due to the instruction RL A, we must ; save it in the stack in order to preserve its value ; that was defined by the main program.</p>
	MOV	TH0, #3Ch	<p>; After the overflow of Timer/Counter 0, ; (TH0) and (TL0) must be reset (become equal ; to zero). Therefore, we must reinitialize them ; with their initial values so that the ; interruption generates the same time between ; interruptions again, otherwise the time between ; interruptions of Timer/Counter 0 changes</p>
	MOV	TH0, #0B0h	
	INC	R7	<p>; (R7) is a counter of 50 ms (it is ; incremented by one unit every 50 ms in the ; service ; subroutine of Timer/Counter 0)</p>
	MOV	A, P1	<p>; It reads (P1) to the content of the ; Accumulator (A or ACC)</p>
	INC	A	<p>; It increments (A) by one unit</p>
	MOV	P1, A	<p>; It copies (A) to (P1)</p>
	POP	PSW	<p>; It recovers (PSW) from the stack (value ; defined by the main program) because it was ; changed in this service subroutine. This</p>

(continued)

			; must be avoided because this would cause a ; malfunction in the control that is being ; performed in the main program.
	POP	ACC	; It recovers (ACC) from the stack (value ; defined by the main program) because it was ; changed in this service subroutine. This ; must be avoided because this would cause a ; malfunction in the control that is being ; performed in the main program.
	RETI		; It returns to the main program
ORG	0070h		; It uses the Assembly guideline: "ORG 0070h" ; or ; "Code at 0070h ... CODE," as described ; above, ; to store the instructions of the service ; subroutine ; of the Timer/Counter 1 interruption (T1INT) in ; 0070h
T1INT:	PUSH	ACC	; As (ACC) will also be used by this service ; subroutine, we must save it in the stack in order ; to preserve its value that was defined by the ; main program.
	PUSH	PSW	; As (PSW) will also be used by this service ; subroutine, we must save it in the stack in order ; to preserve its value that was defined by the ; main program.
	MOV	A, P2	; It reads (P2) to the content of the ; Accumulator (A or ACC)
	DEC	A	; It decrements (A) by one unit
	MOV	P2, A	; It copies (A) to (P2)
	POP	PSW	; It recovers (PSW) from the stack (value ; defined by the main program) because it was ; changed in this service subroutine. This ; must be avoided because this would cause a ; malfunction in the control that is being ; performed in the main program.
	POP	ACC	; It recovers (ACC) from the stack (value ; defined by the main program) because it was ; changed in this service subroutine. This ; must be avoided because this would cause a ; malfunction in the control that is being ; performed in the main program.
	RETI		; It returns to the main program
ORG	0090h		; It uses the Assembly guideline: "ORG 0090h" ; or ; "Code at 0090h ... CODE," as described ; above, ; to store the instructions of the subroutine Sub1 ; in 0090h.
Sub1:	MOV	R0, #4Ch	; Initial address of the memory buffer

(continued)

	MOV	R1, #28h	; Quantity of elements of the memory buffer ; ($=73h - 4Ch + 1 = 28h$)
	MOV	R2, #00h	; (R2) is the counter of elements higher than ; the value F8h and presents bit 3 equal to logic 1
Addr2:	MOV	A, @R0	; It copies the content of the memory buffer ; pointed by (R0) to (A)
	CLR	C	; It clears the carry-flag bit
	SUBB	A, #0F8h	; It subtracts the value F8h from (A)
	JC	Addr1	; If (A) is smaller than F8h, which is the same as ; smaller or equal to F7h, it jumps to Addr1
	MOV	A, @R0	; It copies the content of the memory buffer ; pointed by (R0) to (A)
	JNB	ACC.3, Addr1	; If bit 3 of (A) is equal to logic 0, it jumps ; to Addr1
	INC	R2	; If (A) is higher than the value F7h and ; presents bit 3 equal to logic 1, (R2) is ; incremented by one unit
Addr1:	INC	R0	; It points to next byte of the memory buffer to be ; analyzed
	DJNZ	R1, Addr2	; It decrements (R1) and if it is different from ; zero, it jumps to Addr2 (it has more elements of ; the memory buffer to be analyzed), otherwise ; all ; elements are analyzed
	RET		; It returns from the subroutine to the main ; program
ORG	0120h		; It uses the Assembly guideline: "ORG 0120h" ; or ; "Code at 0120h ... CODE," as described ; above, ; to store the instructions of the subroutine Sub2 ; in 0120h.
Sub2:	MOV	R7, #00h	; It clears (R7) to count the time of 1 s
	MOV	P0, R2	; It increments (A)
Addr3:	MOV	A, R7	; It copies (A) to (R7)
	CLR	C	; It clears the carry-bit flag
	SUBB	A, #14h	; It subtracts 14h from (A) ($=20_{10}$, which ; corresponds to a time of 1 s)
	JC	Addr3	; If (A) < 14h (Is it still not a second?), it waits ; until it reaches 1 ; second. Note: (R7) is incremented every 50 ms ; in the ; service subroutine of Timer/Counter 0 (T0INT)
	MOV	R7, #00h	; It clears (R7) to count the time of 1 s
	MOV	P0, #00	; It clears (P0) (Clear the LEDs)
Addr4:	MOV	A, R7	; It copies (A) to (R7)
	CLR	C	; It clears the carry-bit flag
	SUBB	A, #14h	; It subtracts 14h from (A) ($=20_{10}$, which ; corresponds ; to a time of 1s)

(continued)

	JC	Addr4	; If (A) < 14h (Is it still not a second?), it waits ; until it reaches 1 ; second. Note: (R7) is incremented every 50 ms ; in the ; service subroutine of Timer/Counter 0 (T0INT)
	RET		; It returns to the main program
ORG	0200h		; It uses the Assembly guideline: "ORG 0200h" ; or ; "Code at 0200h ... CODE," as described ; above, ; to store the instructions of the main program in ; 0200h.
MainPr:	MOV	P0, #00h	; It programs/configures port 0 as output
	MOV	P1, #00h	; It programs/configures port 1 as output
	MOV	P2, #00h	; It programs/configures port 2 as output
	MOV	TMOD, #61h	; It programs/configures the Timers/Counters 0 ; and 1 on the modes 1 (M1 M0 = 00 ₂ : counting ; register of 16 bits) and 2 (M1 M0 = 01 ₂ : ; counting ; register of 8 bits with automatic recharge), ; respectively. ; T _{input_clock} of Timer/Counter 0 equal to 1 ; $\mu s = 12/f_{crystal} [(C/\bar{T})=0]$, and turn-on/off by ; software (Gate = 0) ; T _{input_clock} of Timer/Counter 1 equal to 1 ms ; (external clock in T1 pin) [(C/\bar{T})=1] because if ; we ; use f _{crystal} = 12 MHz, the maximum time ; between interruptions is 256 μs , and turn-on/off ; by software (Gate = 0)
	MOV	TH0, #3Ch	; It initializes the counting register of ; Timer/Counter 0
	MOV	TL0, #0B0h	
	MOV	TL1, #38h	; It initializes the counting register of ; Timer/Counter 1
	MOV	TH1, #38h	; It initializes the automatic recharge register of ; Timer/Counter 1
	MOV	IE, #8Ah	; It enables the attending of the Timers/Counters ; 0 ; and 1
	MOV	TCON, #50h	; It turns on the Timers/Counters 0 and 1
Loop:	LCALL	Sub1	; It calls subroutine 1
	LCALL	Sub2	; It calls subroutine 2
	SJMP	Loop	; Loop of the main program
	END		

8.5.1 Proposed Exercises

1. Design a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core microcontroller family which is able to perform the following activities:
 - (a) Program/configure the Timer/Counter 0 to the mode 1. It must generate interruptions every 20 ms.
 - (b) Program/configure the Timer/Counter 1 to the mode 2. It must generate interruptions every 250 μ s.
 - (c) Implement a subroutine that can calculate the quantity of elements which are higher than and equal to 2Fh, smaller than or equal to 8Fh, and present odd parity considering a memory buffer whose initial and final addresses are 3 Ah to 67h, respectively. The result must be put in the content of the port 0 (P0) for 1.5 s (use the Timer/Counter 0 to generate the time of 1.5 s). This subroutine must be stored in the program memory address 0100h.
 - (d) Implement a service subroutine to the interruption source of Timer/Counter 0 that transforms the port 1 (P1) into an ascendant counter of decimal numbers. This subroutine of service must be stored in the program memory address 0200h.
 - (e) Implement a service subroutine to the interruption source of Timer/Counter 1 that transforms the port 2 (P2) into an ascendant counter of even numbers. This subroutine of service must be stored in the program memory address 0030h.
 - (f) The main program must be stored in the program memory address 0400h.
2. Design a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core microcontroller family which is able to perform the following activities:
 - (a) Program/configure the Timer/Counter 0 to the mode 0. It must generate interruptions every 5 ms.
 - (b) Program/configure the Timer/Counter 1 to the mode 2. It must generate interruptions every 500 ms.
 - (c) Implement a subroutine that can calculate the quantity of elements which are smaller than and equal to AFh, and present odd parity regarding a memory buffer whose initial and final addresses are 3Dh to 59h, respectively. The result must be put in the content of the port 0 (P0) for 0.5 s (use Timer/Counter 0 to generate the time of 0.5 s). This subroutine must be stored in the program memory address 0100h.
 - (d) Implement a service subroutine to the interruption source of Timer/Counter 0 that transforms the port 1 (P1) into an ascendant counter of decimal numbers. This subroutine of service must be stored in the program memory address 0200h.
 - (e) Implement a service subroutine to the interruption source of Timer/Counter 1 that transforms the port 2 (P2) into an ascendant counter of

even numbers. This subroutine of service must be stored in the program memory address 0030h.

- (f) The main program must be stored in the program memory address 0400h.
3. Design a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core microcontroller family which is able to perform the following activities:
- (a) Program/configure the Timer/Counter 0 to the mode 2. It must generate interruptions every 1 ms.
 - (b) Program/configure the Timer/Counter 1 to the mode 1. It must generate interruptions every 400 ms.
 - (c) Implement a subroutine that can calculate the quantity of elements which are smaller than and equal to AFh, and present odd parity regarding a memory buffer whose initial and final addresses are 3Dh to 59h, respectively. The result must be put in the content of the port 0 (P0) for 0.5 s (use Timer/Counter 0 to generate the time of 0.5 s). This subroutine must be stored in the program memory address 0100h.
 - (d) Implement a service subroutine to the interruption source of Timer/Counter 0 that rotates the content of the port 1 (P1) of 2 bits to the right. This subroutine of service must be stored in the program memory address 0200h.
 - (e) Implement a service subroutine to the interruption source of Timer/Counter 1 that rotates the content of the port 2 (P2) of 3 bits to the left. This subroutine of service must be stored in the program memory address 0300h.
 - (f) The main program must be stored in the program memory address 0400h.
4. Design a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core microcontroller family which is able to perform the following activities:
- (a) Program/configure the Timer/Counter 0 to the mode 2. It must generate interruptions every 1 ms.
 - (b) Program/configure the Timer/Counter 1 to the mode 1. It must generate interruptions every 400 ms.
 - (c) Implement a subroutine that can calculate the quantity of elements which are smaller than and equal to AFh, and present odd parity considering a memory buffer whose initial and final addresses are 3Dh to 59h, respectively. The result must be put in the content of the port 0 (P0) for 0.5 s (use Timer/Counter 0 to generate the time of 0.5 s). This subroutine must be stored in the program memory address 0100h.
 - (d) Implement a service subroutine to the interruption source of Timer/Counter 0 that flashes the content of the port 1 (P1). This subroutine of service must be stored in the program memory address 0200h.

- (e) Implement a service subroutine to the interruption source of Timer Counter 1 that rotates the content of the port 2 (P2) of 3 bits to the left. This subroutine of service must be stored in the program memory address 0300h.
- (f) The main program must be stored in the program memory address 0400h.

References

1. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
2. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
3. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
4. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
5. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
6. Atmel Corporation (1997) AT89 Series Hardware Description (0499B–B), Dec 1997
7. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
8. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D–MICRO), June 2008
9. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
10. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

The Serial Communication Interface of the 8051 Core Microcontroller

9.1 Introduction

This chapter describes the serial communication interface of the 8051 core microcontroller family, which is responsible for transmitting/receiving serial data to/from computer systems [1–10].

This subject is extremely important because the reader will learn to design software packages that are capable of making serial communication between computer systems [1–10].

Some examples are presented regarding the use of the serial communication interface to transmit and receive serial data to/from other computer systems [1–10].

9.2 The Serial Communication Interface of 8051 Core Microcontrollers

The serial communication is extremely important regarding the intelligent control of computer systems, manufacturing processes, robots, telecommunications, energy systems, etc. [10].

The serial communication is based on human communication. Therefore, consider that two people are talking. To avoid conflicts of the sound signals generated by these two people, so that one person can understand what the other is saying, it is necessary that, while one person speaks through their system of speech, the other is listening through their hearing system, and vice versa, according to Fig. 9.1. We can observe that there is a rule (intrinsic protocol) to make a serial communication between two individuals, which is “while one speaks, the other listens,” and vice versa. This must be respected if there is only one physical location/means to perform the serial communication, i.e., the air (wireless communication) in this case [10].

Therefore, when person 1 speaks to person 2, the speech system of person 1 is the transmitter, the hearing system of person 2 is the receiver, and the physical means is the air, where the information (sound signals) travels in the air (wireless) [10].

Fig. 9.1 Two people performing a serial communication via wireless

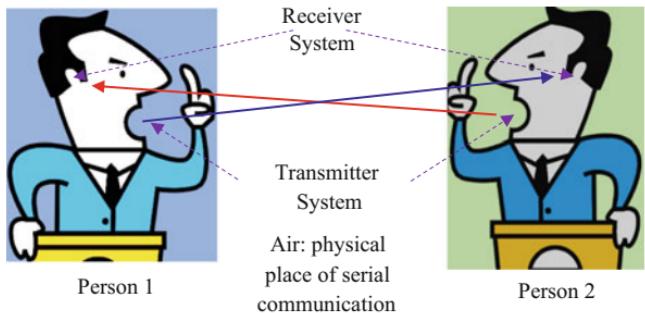
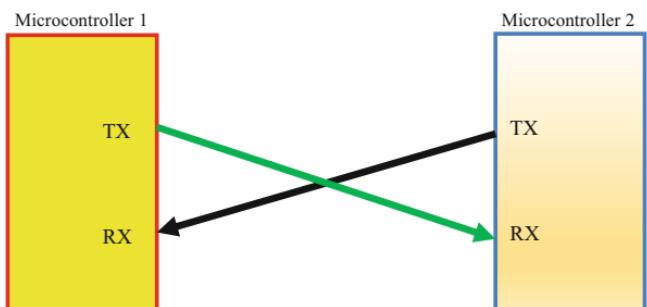


Fig. 9.2 Electrical connections of two microcontrollers performing a serial communication via wireless



It is important to highlight that the communication is serial because each syllable of a word is transmitted/received one by one at a time [10].

Also, observe that the serial communication between these people is asynchronous because it is not based on a transmission/reception of an external clock simultaneously to the serial data which we desire to transmit/receive. The synchronism of an asynchronous serial communication is done by each word transmitted/received, taking into account that there is a pause between each word transmitted/received (absence of sound signals). Consequently, the synchronism of the transmission/reception of information by the two individuals is done by the sound variation between a pause and a transmitted/received word [10].

Therefore, as the serial communication between computer systems are based on human communication, a serial communication between two microcontrollers must be done according to Fig. 9.2, where the transmitter of microcontroller 1 (TX) must be electrically connected to receiver 2 (RX), and transmitter 2 (TX) must be electrically connected to receiver 1 (RX) [10].

As an example of serial communication considers an access control system of employees in a company that has the purpose of storing input and output data (data, time, employee number, etc.) to be later used to generate the staff payroll. In this access control system, the “portable time clock” has the purpose of collecting and storing the input and output information for each employee in its data memory. Besides, the whole access control system also must contain an application software to perform a serial communication between the portable time clock and a microcomputer to collect the information stored in the portable time clock. This information will be used to generate the staff payroll. The collection of this data regarding this

system is usually performed through serial communication of data, which can be done by various means, such as by twisted pair of wire, radio-frequency signals (wireless), microwave, etc. This is just one application among thousands of many others that can be described [10].

Therefore, the study of the serial communication interface is of fundamental relevance for the design of computer systems implemented by microcontrollers [10].

The serial communication interface of the 8051 core microcontroller family is bidirectional (full duplex) and asynchronous. It is bidirectional since it can receive and transmit serial data at the same time and asynchronous because the control bits are incorporated to each byte to be transmitted/received, such as [1–10]:

- *Start bit*: it is responsible for synchronizing the communication between the transmitter and the receiver. The transmitter initially transmits the start bit and then the receiver is synchronized with this electrical signal to start receiving the other bits of data that make up the byte being serially transmitted. Before a byte is transmitted/received, the means of communication is usually in high logic level, and thus the start bit is a baud rate in logic zero.
- *Parity bit* (check bit): besides the start bit and 8 bits (byte) of the data, it is possible to transmit/receive an additional check bit, named “parity bit” (this option can be programmed/configured during the programming/configuring of the serial communication interface). The parity bit is calculated as a function of the byte to be transmitted/received. Initially, the serial communication between two computer systems must be defined if the transmission/reception is performed with even or odd parity. If we consider that that asynchronous serial communication must be 8 bits with even parity to perform a transmission, the transmission subroutine first needs to calculate the parity bit of the byte to be transmitted based on the byte.

This is done as follows:

- If the byte to be transmitted presents an even parity, then the parity bit must be equal to logic 0 taking into account the 9 bits. So, considering the 8 data bits with the parity bit, these 9 bits must present an even parity.
- If the byte to be transmitted presents an even parity, then the parity bit must be equal to logic 1 taking into account the 9 bits. So, regarding the 8 data bits with the parity bit, these 9 bits must present an odd parity.

Regarding the process of receiving serial data, the start bit is discarded, then the 8 bits of the byte that were transmitted are read bit by bit. Afterwards, the parity bit is also read and finally the stop bit (it will explain after) is read, which is discarded too. In this way, the receive interruption source service subroutine must read the 8 bits (byte) that were received, calculate the parity bit, and then compare it with the received one. If the calculated parity, which in this case was considered as even, is equal to that received, then the transmission/reception process was successful.

Otherwise, it means that the received information is different from the transmitted one and consequently the transmission/reception process was not successful.

- **Stop bit:** It is a synchronism and indicates the end of the transmission/reception process. It is usually equal to zero during a baud rate.

Therefore, a byte to be transmitted by the serial communication interface presents: a start bit, the 8 bits (byte) to be transmitted, a parity bit (optional), and a stop bit [1–10].

Another feature of the serial communication interface is that it is buffered, i.e., it can start receiving a second datum (byte) before the previous byte has been read by the receive register (serial buffer, SBUF). However, if the previous byte has not yet been read during the receiving time (baud rate) of the next byte to be read, the previous byte will be lost [1–10].

There are two special function registers that are responsible for transmitting and receiving the data through the serial communication interface, called $SBUF_{out}$ and $SBUF_{in}$, respectively, which are simply called SBUF. When a write operation is performed on the SBUF, it means that a byte will be transmitted through the $SBUF_{out}$. When a read operation is performed with the SBUF, it means that the last byte received by the serial communication will be read from the $SBUF_{in}$ [1–10].

9.3 Programming/Configuring the Serial Communication Interface

The special function register named serial control (SCON) is used to program/configure the operation mode of the serial communication interface [1–10].

bit	7	6	5	4	3	2	1	0
(SCON) =	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SM1	Mode	Description	Baud rate
0	0	0	Shift register	$f_{Crystal}/12$
0	1	1	UART of 8 bits	It is programmed by Timer/Counter 1
1	0	2	UART of 9 bits	$f_{Crystal}/64$ or $f_{Crystal}/32$
1	1	3	UART of 9 bits	It is programmed by Timer/Counter 1

Symbols	Name and description
SM2	It enables the communication between multiple microcontrollers on modes 2 and 3, and if $SM2 = 1$, RI will not be activated if the 9 data bit received is equal to 0. On mode 1, if $SM2 = 1$, RI will not be activated if a valid stop bit is not received. On 0 mode, it should be 0
REN	Set/cleared by the software to enable or disable the serial data reception
TB8	This bit is responsible for transmitting the 9 data bit (usually the parity bit) when the serial communication interface is programmed/configured on modes 2 and 3, respectively. It can be set or cleaned by the software

(continued)

<i>SM0</i>	<i>SM1</i>	<i>Mode</i>	<i>Description</i>	<i>Baud rate</i>
RB8			This bit is responsible for receiving the 9 data bit (usually the parity bit) when the serial communication interface is programmed/configured on modes 2 and 3, respectively. It can be set or cleared by the software. On mode 1, if $SM2 = 0$, RB8 is responsible for receiving the stop bit. On mode 0, RB8 is not used	
TI			It is the transmission interruption flag, set by the hardware at the end of the 8-bit time which is transmitted on mode 0 or at the start of the stop bit regarding the other programming/configuration modes. It indicates that a byte was transmitted serially by the $SBUF_{out}$. It must be cleared by the software to enable a new transmission of another byte, i.e., it is not cleared by the hardware when the service subroutine of the serial communication interruption is attended	
RI			It is the reception interruption flag, set by the hardware at the end of the 8-bit time which is received on mode 0 or in the middle of the time of the stop bit regarding the other programming/configuration modes. It indicates that a byte was received serially by the $SBUF_{in}$. It must be cleared by the software to enable a new reception of another byte, i.e., it is not cleared by the hardware when the service subroutine of the serial communication interruption is attended	

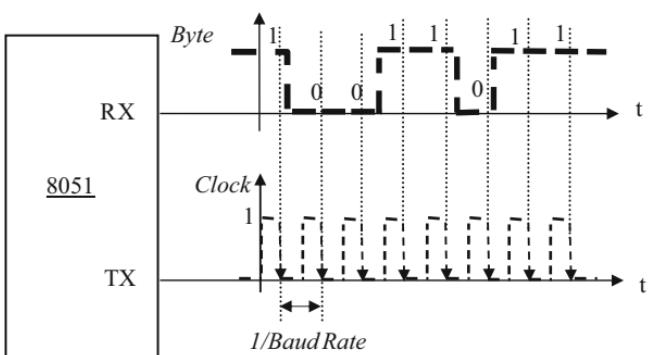
The serial communication interface can operate on four (4) modes [1–10].

9.3.1 Mode 0

In this configuration, the serialized data enter and exit through the RX pin of the 8051 core microcontroller. The byte is transmitted or received as follows: initially the RX pin is on a high logic level and the first bit to be transmitted or received is the least significant bit. On this mode, the TX pin is responsible for transmitting the baud rate synchronized with the byte to be received. Therefore, the receptor reads each bit of the byte taking into account the falling edge of the baud rate transmitted (clock signal) by the TX pin. The baud rate is fixed at 1/12 of the crystal (oscillator) frequency, as shown in Fig. 9.3 [1–10].

The reception is initiated when the reception enabler bit [(REN): bit 4 of the (SCON)] is equal to logic 1, the reception interruption bit (RI) is equal to logic 0, and the alternate function pin P3.1 (RX) is enabled [1–10].

Fig. 9.3 Waveforms of the transmission/reception process of a byte on mode 0



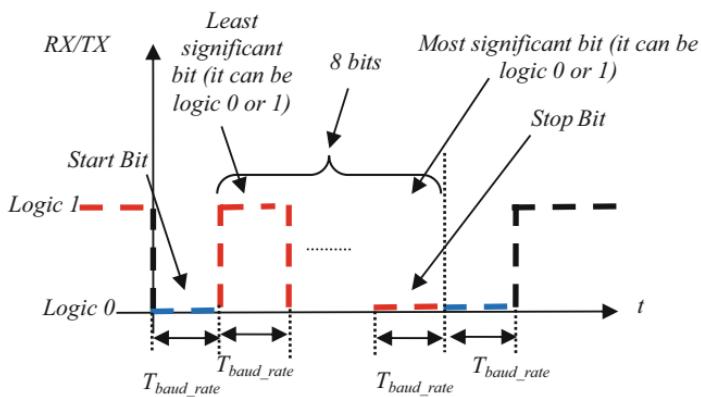


Fig. 9.4 Waveform of the transmission/reception process of a byte on mode 1

When the reception of a datum is finished, the bit reception interrupt [(RI): bit 0 of the (SCON)] is set, indicating that the end of reception process of a byte via serial communication interface was obtained within the reception buffer ($SBUF_{in}$).

The transmission of a datum is initiated by any instruction that uses $SBUF$ as a destination (target) register. In this case, the $SBUF_{out}$ [1–10].

When the most significant bit of the data is in the position to be transmitted, the TI bit is set, indicating that a transmission interruption will occur [1–10].

9.3.2 Mode 1

On mode 1, 10 bits are transmitted through the transmitter pin (TX), and they are received through the receiver pin (RX), which are [1–10]:

- One start bit (logic 0) in a period of the baud rate (T_{baud_rate}) due to the TX pin, which is initially in logic 1
- 8 bits (byte: initially the least significant bit is transmitted by the transmitter or received by the receiver) at every period of the baud rate (T_{baud_rate})
- One stop bit (logic 0) in a period of the baud rate (T_{baud_rate}), which indicates the end of transmission by the transmitter or the end of reception by the receiver, as shown in Fig. 9.4.

In this configuration mode of the serial communication interface, the baud rate is defined by Timer/Counter 1. This Timer/Counter must be programmed/configured to work on mode 2 (counting register of 8 bits with automatic recharge) [1–10].

In the reception, the stop bit is received through the content of bit RB8 [bit 2 of the (SCON)] [1–10].

The reception starts when it detects a falling edge ($1 \rightarrow 0$) in the RX pin of the 8051 core microcontroller, and it is terminated when receiving the stop bit. When the reception is finished, the RI bit [bit 0 of the (SCON)] is set, and it triggers the

attending process of the service subroutine of the interruption of the serial communication interface, which must be stored in the address 0023h. Consequently, if the serial transmission/reception is enabled [(ES): bit 4 of the (IE) and (REN): bit 4 of the SCON], the instruction LCALL 0023h is executed [triggered by the hardware through the (RI)]. This service subroutine is processed until the instruction “return (RET)” is run, which is responsible for returning to the main program [1–10].

The transmission is initiated by any instruction that uses the content of the SBUF as a target register ($SBUF_{out}$), e.g., by the execution of the instruction [MOV SBUF, A \Rightarrow ($SBUF$) \leftarrow (A)]. Therefore, considering every baud rate generated by Timer/Counter 1, one bit is transmitted by the transmitter of the serial communication interface [1–10].

After the transmission of the last data bit (stop bit), the content of the TI bit [bit 1 of the (SCON)] is set, which triggers the process of servicing of the interruption source of the serial communication interface (LCALL 0023h) by the hardware [1–10].

9.3.3 Mode 2

On mode 2, eleven (11) data bits are transmitted through the TX pin and received through the RX pin of the 8051 core microcontroller, which are [1–10]:

- One start bit (logic 0) in a period of the baud rate (T_{baud_rate}) due to the TX pin, which is initially in logic 1
- 8 bits (byte: initially the least significant bit is transmitted by the transmitter or received by the receiver) at every period of the baud rate (T_{baud_rate})
- One 9 bit in a period of the baud rate (T_{baud_rate}), which is usually the byte parity bit that can be even or odd)
- One stop bit (logic 0) in a period of the baud rate (T_{baud_rate}), which indicates the end of transmission by the transmitter or the end of reception by the receiver

The baud rate is programmable to be either 1/64 of the crystal (oscillator) frequency, if the content of the SMOD [bit 7 of the (PCON)] is equal to logic 0, or to be equal to 1/32 of the crystal (oscillator) frequency, and if the content of the serial mode bit [SMOD: bit 7 of the (PCON)] is equal to logic 1 [1–10].

In the reception, the 9 bit of the byte to be received (usually the parity bit) is received through the content of bit RB8 [bit 2 of the (SCON)] of the special function register SCON, while the stop bit is ignored [1–10].

The reception process of a byte is initiated by detecting a falling edge (1 \rightarrow 0) on the RX pin, and it is terminated by detecting the stop bit. As a consequence, the content of the bit “reception interruption (RI)” of the special function register SCON is set. This bit is responsible for triggering the process of servicing the interrupt source of the serial communication interface by running the instruction LCALL 0023h by the hardware. The service subroutine of the reception interrupt source of

the serial communication interface must be stored in the address 0023h (defined by the manufacturer) of the program memory [1–10].

In the transmission, the 9 bit (usually the parity bit) of the byte to be transmitted must be stored in the content of the TB8 bit [bit 3 of the (SCON)]. The objective of the serial transmission/reception with the parity bit is to reduce the errors that can occur during this serial communication if an odd number of data bits is changed [1–10].

The transmission is initiated by any instruction that uses the content of the register “serial buffer (SBUF)” as a target register, and it is terminated when the content of the bit “transmission interruption (TI)” is equal to logic 1 [1–10].

9.3.4 Mode 3

On mode 3, eleven data bits are transmitted through the TX pin by the transmitter and received through the RX pin of the receiver of the 8051 core microcontrollers, which are [1–10]:

- One start bit (logic 0) in a period of the baud rate (T_{baud_rate}) due to the TX pin, which is initially in logic 1
- 8 bits (byte: initially the least significant bit is transmitted by the transmitter or received by the receiver) at every period of the baud rate (T_{baud_rate})
- One 9 bit in a period of the baud rate (T_{baud_rate}), which is usually the byte parity bit that can be even or odd
- One stop bit (logic 0) in a period of the baud rate (T_{baud_rate}), which indicates the end of transmission by the transmitter or the end of reception by the receiver

In fact, mode 3 is similar to mode 2 in all respects, except for the baud rate which is programmable by Timer/Counter 1, programmed/configured to work on mode 2 (counting register of 8 bits with automatic recharge), as described on mode 1.

The serial communication between multiprocessors is performed through modes 2 and 3. On these modes, 9 bits of data are received by the receiver, in which the 9 bit is stored in the content of bit RB8 [bit 2 of the (SCON)]. This serial communication is terminated when the stop bit is received [1–10].

The serial communication interface can be programmed/configured to be terminated when the stop bit is received; thus, the serial reception interrupt bit “reception interruption (RI)” is activated only when the bit (RB8) is equal to 1. This feature is enabled by setting the SM2 bit in SCON. One way to use this feature in multiprocessor systems is when a master processor wants to transmit a block of data to one of the slaves, in which it first sends a byte related to the address of the slave [1–10].

An address byte of a slave differs from one byte of data. When the 9 bit is equal to logic 1, it means that it is a byte of address of a slave, and when the 9 bit is equal to logic 0, it means that it is a byte of data [1–10].

When the content of the SM2 [bit 5 of the (SCON)] is equal to logic 1, no slave will be interrupted by one byte of data. A byte of address, however, will disrupt all slaves, and each slave will examine the received byte and verify if it is being addressed. The addressed slave will clear the contents of its SM2 bit and prepare to receive the bytes of data that will be transmitted by the master. Slaves that have not been addressed leave their SM2 bit set and ignore the bytes of data that will be sent by the master [1–10].

The content of the SM2 has no effect on mode 0, and on mode 1, it can be used to check the validity of the stop bit [1–10].

Regarding the reception on mode 1, if the content of the SM2 is equal to logic 1, the reception interruption will not be activated, unless a valid stop bit is received [1–10].

The baud rate on mode 0 is fixed and equal to $f_{\text{crystal}}/12$. On mode 1, it depends on the content of the bit “serial mode (SMOD)” [bit 7 of the (PCON)]. If content of the SMOD is equal to logic 0 (default value after a reset signal), the baud rate is equal to 1/64 of the crystal (oscillator) frequency (f_{crystal}). If the content of the SMOD is equal to logic 1, the baud rate is equal to 1/32 of the crystal (oscillator) frequency (f_{crystal}). On modes 1 and 3, the baud rate is determined by the overflow 1111 ... 11110000 ... 0000 of Timer/Counter 1. The baud rate of Timer/Counter 1 can be calculated by Eq. (9.1) [1–10].

$$\text{Baud Rate} = 2^{\text{SMOD } (0 \text{ or } 1)} / 32 \cdot (\text{Timer/Counter 1 overflow rate}) \text{ (bits/s)} \quad (9.1)$$

The interruption of Timer/Counter 1 must be disabled for this application. In most applications, it is configured to operate as a Timer/Counter in the automatic recharge mode (upper nibble of the content of the TMOD equal to 0010_2). In this case, the baud rate is given by Eq. (9.2) [1–10].

$$\text{Baud Rate} = 2^{\text{SMOD } (0 \text{ or } 1)} / 32 \cdot (f_{\text{crystal}} / \{12 \cdot [256 - (\text{TH1})]\}) \text{ (bits/s)} \quad (9.2)$$

Table 9.1 illustrates various configurations of possible baud rates that can be programmed/configured for the serial communication interface. For each baud rate, it is necessary to set the crystal frequency (f_{crystal}), the content of the bit SMOD, and whether Timer/Counter 1 programmed on mode 2 will be used to define the baud rate. To illustrate, consider that it is desired to perform serial communication between two microcontrollers with a baud rate of 9.6 Kbits/s. To do this, it is necessary to use a crystal (Xtal) of 11.059 MHz to define the content of the SMOD equal to logic 0 and to program/configure Timer/Counter 1 on mode 2 (counting register of 8 bits with automatic recharge), in which the initial value of the counting and recharge registers must be equal to FDh, as indicated in Table 9.1 [1–10].

Table 9.1 Programming the baud rate through Timer/Counter 1

Timer/Counter 1					
Baud rate (bits/s)	Crystal (oscillator) frequency (MHz)	SMOD	C/T	Configuration of the operation mode of Timer/Counter 1	Value to be stored in the counting and recharge registers of Timer/Counter 1 which must be programmed on operation mode 2
Mode 0 max: 1 MHz	12	X	X	X	X
Mod2 2 max: 375 K	12	1	X	X	X
Mode 1 and 3: 62,5 K	12	1	0	2	FF _h
19,2 K	11,059	1	0	2	FD _h
9,6 K	11,059	0	0	2	FD _h
4,8 K	11,059	0	0	2	FA _h
2,4 K	11,059	0	0	2	F4 _h
1,2 K	11,059	0	0	2	E8 _h
137,5	11,059	0	0	2	1D _h
110	6	0	0	2	72 _h

9.4 Beginning and Ending the Transmission and Reception of a Byte Through the Serial Communication Interface

When a datum is stored in the special function register SBUF, e.g., MOV SBUF, A [(SBUF)←(A)], which in this case is the SBUF_{out}, this byte is stored in the transmission buffer, and the serial transmission process is initiated [1–10].

When a byte is copied from the content of the SBUF to a certain content of a register or a memory location (e.g., MOV A, SBUF), it comes from the receive buffer, which is the SBUF_{in}, i.e., it reads one byte of the serial communication interface [1–10].

On mode 0, the reception is initialized by the condition of the contents of the RI bits [bit 0 of the (SCON)] equal to logic 0 and REN [bit 4 of the (SCON)] equal to logic 1 [1–10].

On the other operating modes (1 to 3), the reception is initiated by the arrival of the start bit, if the content of the REN bit [bit 4 of the (SCON)] is equal to logic 1 [1–10].

On all modes, the serial transmission is initialized by any instruction that uses the content of the SBUF as the destination register [1–10].

When receiving the last bit (most significant bit) of a byte received by the serial communication interface, the content of the RI bit [bit 0 of the (SCON)] is set,

signaling that one byte has been received, and it is available to be read on the SBUF_{in} . The reception interruption bit (RI) triggers the execution of the instruction LCALL 0023h, if the serial communication interface [(ES): bit 4 of the interruption enable (IE) is equal to 1] and the data reception (REN) [bit 4 of the (SCON) is equal to 1] are enabled. Therefore, the service subroutine of the serial communication interface (transmission/reception) is addressed to the program memory address 0023h. In this address, the service subroutine of the serial communication interface must be stored in the program memory so that the received byte can be read and decoded and trigger some actions needed to execute the machine/process control. Generally, the reception interruption flag [(RI): bit 0 of the (SCON)] must be reset by the service subroutine of the interruption source of the serial communication interface in order to release the serial communication interface for reception of a new byte [1–10].

After the transmission of the last bit of the data (most significant bit), the content of transmission interruption flag [(TI): bit 1 of the (SCON)] is set, indicating that the transmission has been terminated. Consequently, the interruption of the serial communication interface is enabled [(ES): bit 4 of the (IE) is equal to logic 1] and the TI flag triggers the execution of the instruction LCALL 0023h by the hardware, where the service subroutine of the transmission must be stored in the program memory (same location that the service subroutine must be stored). Commonly, the TI flag must be reset in the service subroutine of the transmission of the serial communication interface in order to release a new data transmission [1–10].

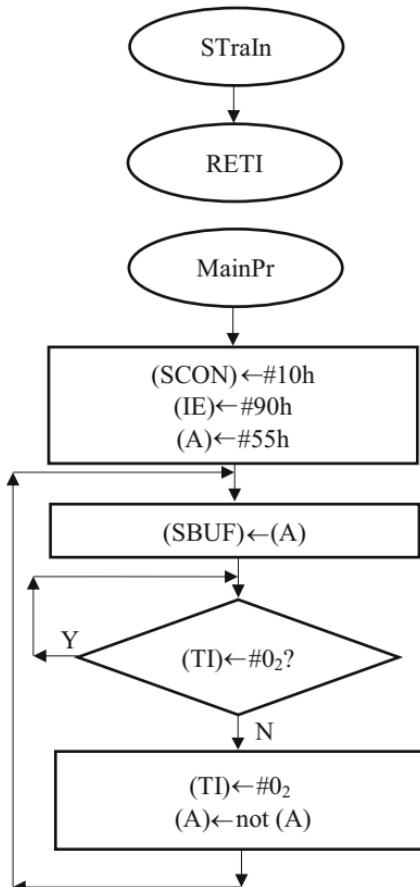
If the reception and transmission are enabled at the same time, the service subroutine of the serial communication interface must distinguish which one was the interruption source that caused the interruption to the CPU. This is done by testing the contents of the RI and TI flags if they are equal to 1 [1–10].

9.5 Solved Exercises

9.5.1 Design a structured program (flowchart and source program) in Assembly that uses one of the members of the 8051 core microcontroller family, which uses the serial communication interface to transmit data. The characteristics of this communication are:

- The serial communication interface must be programmed on the mode 0.
- It must transmit the values 55h and AAh continuously.
- The main program must be written in the program memory address 0100h.

Solution



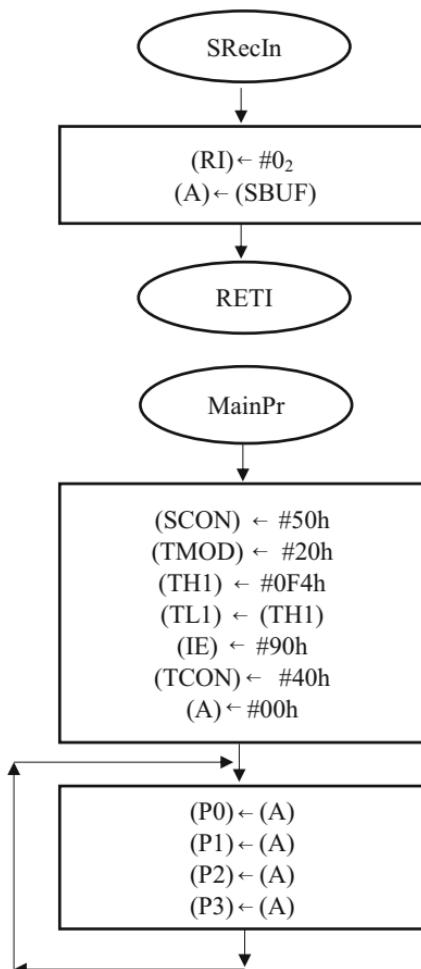
Address of the memory program	Mnemonic	Argument	Comments
ORG	0000h		; It uses the Assembly guideline: "ORG ; 0000h" or "Code at ; 0000h" to write the instruction LJMP ; 0100h in the address ; 0000h. It must be written in the first ; column: ; ORG 0000h ; LJMP 0100h ; Or ; Code at 0000h ; LJMP 0100h ; Code
0000h	LJMP	0100h	; It jumps to the MainPr, whose address is ; at 0100h

(continued)

Address of the memory program	Mnemonic	Argument	Comments
ORG	0023h		; It uses the Assembly guideline: “ORG ; 0023h” or “Code at ; 0023h . . . Code” to write the RETI instruction in ; the address 0023h
STrIn:	RETI		; it returns to the main program
ORG	0100h		; It uses the Assembly guideline: “ORG ; 0100h” or “Code at ; 0100h . . . CODE,” to write the ; instructions of the main ; program in the program memory address ; subroutine ; 0100h
MainPr:	MOV	SCON, #10h	; It programs/configures the serial ; communication interface ; to work on mode 0 (shift register)
	MOV	IE, #90h	; It enables the attending of the service ; subroutine of the ; transmission of the serial communication ; interface
	MOV	A, #55h	; It initializes (A) with the value of 55h
Loop:	MOV	SBUF, A	; It copies (A) to the (SBUF _{out}) and ; consequently the ; transmission is initialized
	JNB	TI, \$; Has not the byte been transmitted yet? If ; no, it waits for it ; to be transmitted
	CLR	TI	; It resets the (TI) to release for a new serial ; transmission
	CPL	A	; It complements (A): 55h ⇔ AAh
	SJMP	Loop	; It jumps to the program memory address ; loop
	END		

9.5.2 Design a structured program (flowchart and source program) in Assembly that uses one of the members of the 8051 core microcontroller family. It receives data through the serial communication channel and makes them available on all output ports (P0 to P3), which are connected to four sets of eight LEDs. The characteristics of this serial communication are:

- The serial communication channel must be programmed on mode 1 with a baud rate of 2.4 kHz.
- The main program must be written in the program memory address 0700h.



Address of the memory program	Mnemonic	Argument	Comments
ORG	0000h		; It uses the Assembly guideline: “ORG ; 0000h” or “Code at ; 0000h” to write the instruction LJMP ; 0100h in the address ; 0000h. It must be written in the first ; column: ; ORG 0000h ; LJMP 0100h ; Or ; Code at 0000h ; LJMP 0100h ; Code
0000h	LJMP	0100h	; It jumps to the MainPr, whose address is ; at 0100h

(continued)

Address of the memory program	Mnemonic	Argument	Comments
ORG	0023h		; It uses the Assembly guideline: “ORG ; 0023h” or “Code at ; 0023h . . . Code” to store the instructions ; of the service ; subroutine of the reception interruption in ; the address ; 0023h
STraIn:	CLR	RI	; it resets the (RI) to release for a new serial ; reception
	MOV	A, SBUF	; it copies the (SBUF) to (A) (it reads the ; reception ; buffer after the receiving of the byte in ; the serial ; communication interface)
	RETI		; it returns to the main program
ORG	0100h		; It uses the Assembly guideline: “ORG ; 0100h” or “Code at ; 0100h . . . CODE” to write the ; instructions of the main ; program in the program memory address ; subroutine ; 0100h
MainPr:	MOV	SCON, #50h	; It programs/configures the serial ; communication interface ; to work on mode 1 (UART of 8 bits and ; the baud rate ; programmed by Timer/Counter 1)
	MOV	TMOD, #20h	; It programs/configures Timer/Counter ; 1 on mode 2 to ; be used as a baud rate generator
	MOV	TL1, #0F4h	; It determinates a baud rate of ; 2,4 Kbits/s (KHz). To do ; this, it is necessary to initialize the (TL1), ; which is the ; counting register, with the value of F4h. ; Note that it is ; necessary to use a crystal of 11,052 MHz, ; according to ; Table 9.1
	MOV	TH1, TL1	; It initializes the automatic recharge ; register (TH1) with the ; value of F4 too
	MOV	IE, #90h	; It enables the attending of the service ; subroutine of the ; reception of the serial communication ; interface
	MOV	TCON, #40h	; It turns on Timer/Counter 1 to start ; generating the ; baud rate
	MOV	A, #00h	; It initializes (A) with zero

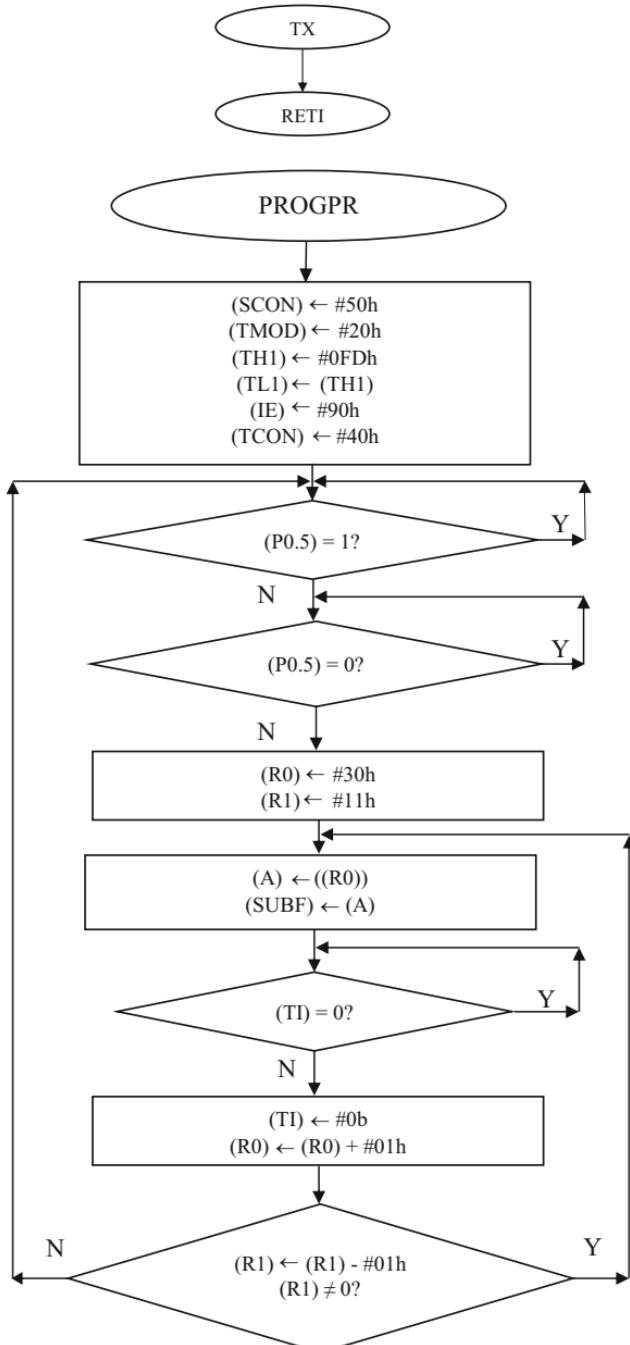
(continued)

Address of the memory program	Mnemonic	Argument	Comments
Loop:	MOV	P0, A	; After receiving a byte from reception ; buffer, it copies ; (A) = (SBUF _{in}) to the (P0)
	MOV	P1, A	; After receiving a byte from reception ; buffer, it copies ; (A) = (SBUF _{in}) to the (P1)
	MOV	P2, A	; After receiving a byte from reception ; buffer, it copies ; (A) = (SBUF _{in}) to the (P2)
	MOV	P3, A	; After receiving a byte from reception ; buffer, it copies ; (A) = (SBUF _{in}) to the (P3)
	SJMP	Loop	; It jumps to the program memory address ; loop. Every time ; new data are received, the contents of the ; ports are ; updated with the received byte from the ; reception buffer ; (SBUF _{in})
	END		

9.5.3 Implement a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core microcontroller family that is able to perform the following activities:

- Program/configure the serial communication interface on mode 1 with a baud rate of 9.6 kHz.
- Transmit the contents of the memory buffer whose initial and final addresses are 30h and 40h, respectively, every time the key 5 of a dipswitch of eight keys (these keys are numbered from 0 to 7, electrically connected to the port 0 and disregard the bounce) is triggered and deactivated.
- The main program must be written in the program memory address 0200h.

Solution



Address of the memory program	Mnemonic	Argument	Comments
ORG	0000h		; It uses the Assembly guideline: “ORG ; 0000h” or “Code at ; 0000h” to write the instruction LJMP ; 0200h in the address ; 0000h. It must be written in the first ; column: ; ORG 0000h ; LJMP 0200h ; Or ; Code at 0000h ; LJMP 0200h ; Code
0000h	LJMP	0200h	; It jumps to the MainPr, whose address is ; at 0100h
ORG	0023h		; It uses the Assembly guideline: “ORG ; 0023h” or “Code at ; 0023h . . . Code” to write the RETI ; instruction in ; the address 0023h
SRecIn:	STraIn:	RETI	; it returns to the main program
ORG	0200h		; It uses the Assembly guideline: “ORG ; 0200h” or “Code at ; 0200h . . . CODE” to store the ; instructions of the main ; program in the program memory address ; 0200h
MainPr:	MOV	SCON, #50h	; It programs/configures the serial ; communication interface ; to work on mode 1 (UART of 8 bits and ; the baud rate ; programmed by Timer/Counter 1)
	MOV	TMOD, #20h	; It programs/configures Timer/Counter ; 1 on mode 2 to ; be used as a baud rate generator
	MOV	TL1, #0FDh	; It determinates a baud rate of 9.6 Kbits/s ; (kHz). To do ; this, it is necessary to initialize the (TL1), ; which is the ; counting register, with the value of FDh. ; Note that it is ; necessary to use a crystal of 11,052 MHz, ; according to ; Table 9.1
	MOV	TH1, TL1	; It initializes the automatic recharge ; register (TH1) with the ; value of FD, too
	MOV	IE, #90h	; It enables the attending of the service ; subroutine of the ; transmission of the serial communication ; interface
	MOV	TCON, #40h	; It turns on Timer/Counter 1 to start ; generating the

Address of the memory program	Mnemonic	Argument	Comments
Loop:	JNB	P0.5, \$; It waits for key 5 of the dipswitch to be triggered
	JB	P0.5, \$; It waits for key 5 of the dipswitch to be deactivated
	MOV	R0, #30h	; It initializes the (R0) with the values of the initial address of the memory buffer
	MOV	R1, #11h	; It initializes the (R1) with the quantity of the elements of the memory buffer (=40h - 30h + 1 = 11h) to be analyzed
Addr1:	MOV	A, @R0	; It copies the content of a byte of the memory buffer to (A)
	MOV	SBUF, A	; It copies the content of (A) to the (SBUF _{out}) and at this time the serial transmission is started
	JNB	TI, \$; Has not the byte been transmitted yet? If no, it waits for it to be transmitted
	CLR	TI	; It resets the (TI) to release for a new serial transmission
	INC	R0	; It points to the address of the next byte of the memory buffer to be transmitted
	DJNZ	R1, Addr1	; It decrements the (R1) by one unit and verifies if it is different from zero. If it is different from zero, it means that there are more bytes in the memory buffer to be transmitted and therefore, it jumps to the program memory address Addr1
	SJMP	Loop	; It jumps to the program memory address Loop for new transmission of the contents of the memory buffer
	END		

9.6 Proposed Exercises

- 9.6.1 Design a structured program (flowchart and source program) in Assembly that uses one of the members of the 8051 core microcontroller family, which uses the serial communication interface to transmit data. The characteristics of this communication are:
- The serial communication interface must be programmed on mode 2 (baud rate of $f_{\text{Crystal}}/64$ and even parity).
 - It must transmit the following bytes: 01h, 02h, 03h, and 04h continuously.
- 9.6.2 Design a structured program (flowchart and source program) in Assembly that uses one of the members of the 8051 core microcontroller family. It receives data through the serial communication channel and makes them available on all output ports (P0 to P3), which are connected to four sets of eight LEDs. The characteristics of this serial communication are:
- The serial communication channel must be programmed on mode 3 (baud rate of 1.2 kHz and odd parity)
- 9.6.3 Implement a structured program (flowchart and source program) in Assembly using one of the members of the 8051 core microcontroller family that can perform the following activities:
- To program/configure the serial communication interface on mode 1 with a baud rate of 2.4 kHz.
 - The received data must be stored in a circular memory buffer whose initial and final addresses are 38h and 64h, respectively, every time the key 2 and 6 of a dipswitch of eight keys (these keys are numbered from 0 to 7, electrically connected to the port 1 and disregard the bounce) is triggered and deactivated.
 - The main program must be written in the program memory address 0200h.

Note: Circular memory buffer: once the memory buffer is totally used (all addresses were used to store data), we must reinitialize it (point to the initial address again) to allow a new storing of data on it from its initial address.

References

1. Intel Corporation (1994) MCS 51 Microcontroller Family User's Manual (order number 272383-002), Feb 1994
2. Intel Corporation (1980) Using the Intel MCS-51 Boolean Processing Capabilities, Application note (AP-70), Apr 1980
3. Intel Corporation (1996) 8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual (John Wharton, Microcontroller Application), May 1996
4. Philips Semiconductors (1997) 80C51 family programmer's guide and instruction set, Sep 1997
5. Infineon Technologies (2000) C500 – Architecture and Instruction Set – Microcontrollers – User's Manual, July 2000
6. Atmel Corporation (1997) AT89 Series Hardware Description (0499B–B), Dec 1997

7. Atmel Corporation (2001) 8-bit Microcontroller with 4K Bytes In-System Programmable Flash (Rev. 2487A), Oct 2001
8. Atmel Corporation (2008) 8-bit Microcontroller with 32K Bytes Flash (AT89C51RC – 1920D–MICRO), June 2008
9. Texas Instruments (2014) “CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee® Applications”; “CC2540/41 System-on-Chip Solution for 2.4- GHz Bluetooth® low energy Applications – User Guide”, Literature Number: SWRU191F, April 2009–Revised Apr 2014
10. Gimenez SP (2010) Microcontroladores 8051 – Teoria e Prática Editora Érica

8051 Instructions Set

Data transfer operations				opcode			
Instruction	Byte	Cycles	Code	Symbolic representation			
Movement within the microcontroller (no flag is affected)							
The content of the second register/memory is copied to the content of the first register/memory							
MOV A,Rn	1	1	1110 1rrr	(PC) \leftarrow (PC) + 1 (A) \leftarrow (Rn)	A AC ACC		
			1110 1000 1110 1111		The program counter is added by one unit, and the content of the Rn register is copied to the content of the accumulator (A) register. No flag is affected		
MOV A, direct	2	1	1110 0101 direct address	(PC) \leftarrow (PC) + 2 (A) \leftarrow (direct)	The program counter is added by two units, and the content of the position of the internal RAM whose address is direct is copied to the content of the accumulator (A) register		
MOV A, @Ri	1	1	1110 0111	(PC) \leftarrow (PC) + 1 (A) \leftarrow ((Ri))	The program counter is added by one unit, and the content of the memory location whose address is given by the content of the Ri register is copied to the content of the accumulator (A) register		
MOV A, #data	2	1	0111 0100 immediate data	(PC) \leftarrow (PC) + 2 (A) \leftarrow #data	The program counter is added by two units, and the date value is copied to the content of the accumulator (A) register		

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
MOV Rn,A	1	1	1111 1rrr	$(PC) \leftarrow (PC) + 1$ $(Rn) \leftarrow (A)$	The program counter is added by one unit, and the content of the accumulator (A) register is copied to the content of the Rn register
MOV Rn, direct	2	2	1010 1rrr direct address	$(PC) \leftarrow (PC) + 2$ $(Rn) \leftarrow (\text{direct})$	The program counter is added by two units, and the content of the internal RAM location whose address is direct is copied to the content of the Rn register
MOV Rn, #data	2	1	0111 1rrr immediate data	$(PC) \leftarrow (PC) + 2$ $(Rn) \leftarrow \#data$	The program counter is added by two units, and the date value is copied to the content of Rn register
MOV direct, A	2	1	1111 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(\text{direct}) \leftarrow (A)$	The program counter is added by two units, and the content of the accumulator (A) register is copied to the content of the internal RAM location whose address is direct
MOV direct, Rn	2	2	1000 1rrr direct address	$(PC) \leftarrow (PC) + 2$ $(\text{direct}) \leftarrow (Rn)$	The program counter is added by two units, and the content of the Rn register is copied to the content of the internal RAM memory location whose address is direct
MOV direct, direct	3	2	1000 0101 dir.adr.src dir.adr. dest	$(PC) \leftarrow (PC) + 3$ $(\text{direct}) \leftarrow (\text{direct})$	The program counter is added by three units, and the content of the position of the internal RAM whose address is direct is copied to the content of the internal RAM memory location whose address is direct
MOV direct, @Ri	2	2	1000 011i direct address	$(PC) \leftarrow (PC) + 2$ $(\text{direct}) \leftarrow ((Ri))$	The program counter is added by two units, and the content of the internal RAM position whose address is given

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
					by the content of the Ri register is copied to the content of the internal RAM memory location whose address is direct
MOV direct, #data	3	2	0111 0101 dir. adr. immed. data	(PC) \leftarrow (PC) + 3 (direct) \leftarrow #data	The program counter is added by three units, and the date value is copied to the content of the internal RAM location whose address is direct
MOV @Ri, A	1	1	1111 011i	(PC) \leftarrow (PC) + 1 ((Ri)) \leftarrow (A)	The program counter is added by one unit, and the content of the accumulator (A) register is copied to the content of the internal RAM position whose address is given by the content of the Ri register
MOV @Ri, direct	2	2	1010 011i direct address	(PC) \leftarrow (PC) + 2 ((Ri)) \leftarrow (direct)	The program counter is added by two units, and the content of the internal RAM location whose address is direct is copied to the content of the internal RAM position whose address is given by the content of the Ri register
MOV @Ri, #data	2	1	0111 011i immediate data	(PC) \leftarrow (PC) + 2 ((Ri)) \leftarrow #data	The program counter is added by two units, and the content of the internal RAM position whose address is given by the content of the Ri register is initialized with the date value
MOV DPTR, #data ₁₆	3	2	1001 0000 imme. data _{15,8} imme. data _{7,0}	(PC) \leftarrow (PC) + 3 (DPH) \leftarrow #data _{15,8} (DPL) \leftarrow #data _{7,0}	The program counter is added by three units, and the value data ₁₆ is copied to the content of the 16-bit DPTR register

data pointer

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Moving the content from the program memory to the content of the accumulator (A) register (no flag is affected)					
MOVC A, @A + DPTR	1	2	1001 0011	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow ((A) + (DPTR))$	The program counter is added by one unit, and the content of the program memory position addressed by the sum of the contents of the accumulator (A) and DPTR registers is copied to the content of the accumulator (A) register
MOVC A, @A + PC	1	2	1000 0011	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow ((A) + (PC))$	The program counter is added by one unit, and the content of the program memory position addressed by the sum of the contents of the accumulator (A) and PC registers is copied to the content of the accumulator (A) register
Moving the content from the external data memory to the content of the accumulator (A) register and vice versa (no flag is affected)					
MOVX A, @Ri	1	2	1110 001i	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow ((Ri))$	The program counter is added by one unit, and the content of the memory location of the external RAM whose address is given by the content of the Ri register is copied to the content of the accumulator (A) register
MOVX A, @DPTR	1	2	1110 0000	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow ((DPTR))$	The program counter is added by one unit, and the content of the memory location of the external RAM whose address is given by the content of the DPTR register is copied to the content of the accumulator (A) register

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
MOVX @Ri,A	1	2	1111 000i	$(PC) \leftarrow (PC) + 1$ $((Ri)) \leftarrow (A)$	The program counter is added by one unit, and the content of the accumulator (A) register is copied to the content of the memory location of the external RAM whose address is given by the content of the Ri register
MOVX @DPTR,A	1	2	1111 0000	$(PC) \leftarrow (PC) + 1$ $((DPTR)) \leftarrow (A)$	The program counter is added by one unit, and the accumulator (A) register is copied to the content of the memory location of the external RAM whose address is given by the content of the D PTR register

Storing data with direct addressing in the stack (no flag is affected)

PUSH direct	2	2	1100 0000 direct address	$(PC) \leftarrow (PC) + 2$ $(SP) \leftarrow (SP) + 1$ $((SP)) \leftarrow (\text{direct})$	The program counter is added by two units, the content of the stack pointer (SP) register is added by one unit (it points to the next memory location), and the content of the register/memory location <src> is copied to the content of the memory location whose address is given by the contents of the SP register
-------------	---	---	--------------------------------	-------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Reading data from the stack to the content of data memory with direct addressing (no flag is affected)

POP direct	2	2	1101 0000 direct address	$(PC) \leftarrow (PC) + 2$ $(\text{direct}) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$	The program counter is added by two units, the content of the memory location whose address is given by the content of the SP register is copied to the content of the register/memory location <dest>, and the content of the stack pointer (SP) register is
------------	---	---	--------------------------------	-------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
					decremented by one unit (it points to the memory position of the previously stored data in the stack)
Data exchange (no flag is affected)					
XCH A,Rn	1	1	1100 1rrr	$(PC) \leftarrow (PC) + 1$ $(A) \leftrightarrow (Rn)$	The program counter is added by one unit and exchanges the content of the accumulator (A) register with the content of the Rn register
XCH A, direct	2	1	1100 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(A) \leftrightarrow (\text{direct})$	The program counter is added by two units and exchanges the content of the accumulator (A) register with the content of the position of internal RAM whose address is direct
XCH A,@Ri	1	1	1100 011i	$(PC) \leftarrow (PC) + 1$ $(A) \leftrightarrow ((Rn))$	The program counter is added by one unit and exchanges the content of the accumulator (A) register with the content of the internal RAM position whose address is given by the content of the Ri register
XCHD A, @Ri	1	1	1101 011i	$(PC) \leftarrow (PC) + 1$ $(A_{3-0}) \leftrightarrow ((Rn_{3-0}))$	The program counter is added by one unit and exchanges the content of the least significant 4 bits of the accumulator (A) register with the content of the least significant 4 bits of the content of the internal RAM position whose address is given by the contents of the Ri register

Arithmetic operations

Addition (all flags are affected)

ADD A,Rn	1	1	0010 1rrr	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A) + (Rn)$	The program counter is added by one unit, and
----------	---	---	-----------	-----------------------------------------------------------	-----------------------------------------------

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
					the result of the addition operation between the contents of the accumulator (A) register and Rn register is stored in the content of the accumulator (A) register
ADD A, direct	2	1	0010 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow$ $(A) + (\text{direct})$	The program counter is added by two units, and the result of the addition operation between the contents of the accumulator (A) register and the position of the internal RAM whose address is direct is stored in the content of the accumulator (A) register
ADD A,@Ri	1	1	0010 0111	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A) + ((Ri))$	The program counter is added by one unit, and the result of the addition operation between the contents of the accumulator (A) register and the position of the internal RAM whose address is given by the content of the Ri register is stored in the content of the accumulator (A) register
ADD A, #data	2	1	0010 0100 immediate data	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow (A) + \#data$	The program counter is added by two units, and the result of the addition operation between the contents of the accumulator (A) register and the date value is stored in the content of the accumulator (A)

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Addition with carry-bit (all flags are affected)					
ADDC A,Rn	1	1	0011 1rrr	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow$ $(A) + (C) + (Rn)$	The program counter is added by one unit, and the result of the addition operation among the contents of the accumulator, the carry-bit, and the Rn register is stored in the content of the accumulator (A) register
ADDC A, direct	2	1	0011 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow$ $(A) + (C) + (\text{direct})$	The program counter is added by two units, and the result of the addition operation among the contents of the accumulator, the carry-bit flag, and the position of the internal RAM whose address is direct is stored in the content of the accumulator (A) register
ADDC A, @Ri	1	1	0011 011I	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow$ $(A) + (C) + ((Ri))$	The program counter is added by one unit, and the result of the addition operation among the contents of the accumulator, the carry-bit flag, and the position of the internal RAM whose address is given by the content of the Ri recorder is stored in the content of the accumulator (A) register
ADDC A, #data	2	1	0011 0100 immediate data	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow$ $(A) + (C) + \#data$	The program counter is added by two units, and the result of the addition operation among the contents of the accumulator, carry-bit flag, and date value is stored in the content of the accumulator (A) register

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Subtraction with carry-bit (all flags are affected)					
SUBB A,Rn	1	1	1001 1rrr	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A)-(C)-$ (Rn)	The program counter is added by one unit, and the result of the subtraction operation among the contents of the accumulator, carry-bit flag, and Rn register is stored in the content of the accumulator (A) register
SUBB A, direct	2	1	1001 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow (A)-(C)-$ $(direct)$	The program counter is added by two units, and the result of the subtraction operation among the contents of the accumulator, carry-bit flag, and position of internal RAM whose address is direct is stored in the content of the accumulator (A) register
SUBB A, @Ri	1	1	1001 010i	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A)-(C)-$ $((Ri))$	The program counter is added by one unit, and the result of the subtraction operation among the contents of the accumulator, the carry-bit flag, and the position of internal RAM whose address is given by the content of the Ri recorder is stored in the content of the accumulator (A) register
SUBB A, #data	2	1	1001 0100 immediate data	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow (A)-(C)-$ $\#data$	The program counter is added by two units, and the result of the subtraction operation among the contents of the accumulator, the carry-bit flag, and the date value is stored in the contents of the accumulator (A) register

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Adding one unit (increment) [the (C), (AC), and (OV) are not affected]					
INC A	1	1	0000 0100	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A) + 1$	The program counter is added by one unit, and the content of the accumulator (A) register is incremented by one unit
INC Rn	1	1	0000 1rrr	$(PC) \leftarrow (PC) + 1$ $(Rn) \leftarrow (Rn) + 1$	The program counter is added by one unit, and the content of the Rn register is incremented by one unit
INC direct	2	1	0000 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(direct) \leftarrow (ditect) + 1$	The program counter is added by two units, and the content of the internal RAM position whose address is direct is incremented by one unit
INC @Ri	1	1	0000 011i	$(PC) \leftarrow (PC) + 1$ $((Ri)) \leftarrow ((Ri)) + 1$	The program counter is added by one unit, and the content of the internal RAM position, whose address is given by the content of the Ri register, is incremented by one unit
INC DPTR	1	2	1010 0011	$(PC) \leftarrow (PC) + 1$ $(DPTR) \leftarrow (DPTR) + 1$	The program counter is added by one unit, and the content of the DPTR register is incremented by one unit
Subtraction of a unit (decrement) [the (C), (AC), and (OV) are not affected]					
DEC A	1	1	0001 0010	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A) - 1$	The program counter is added by one unit, and the content of the accumulator (A) register is decremented by one unit
DEC Rn	1	1	0001 1rrr	$(PC) \leftarrow (PC) + 1$ $(Rn) \leftarrow (Rn) - 1$	The program counter is added by one unit, and the content of the Rn register is decremented by one unit

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
DEC direct	2	1	0001 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(direct) \leftarrow (direct) - 1$	The program counter is added by two units, and the content of the internal RAM location, whose address is direct, is decremented by one unit
DEC @Ri	1	1	0001 0111i	$(PC) \leftarrow (PC) + 1$ $((Ri)) \leftarrow ((Ri)) - 1$	The program counter is added by one unit, and the content of the internal RAM position, whose address is given by the content of the Ri register, is decremented by one unit

Multiplication [all flags are affected, except the auxiliary carry-bit (AC)]

MUL AB	1	4	1010 0100	$(PC) \leftarrow (PC) + 1$ $result_{15-0}$ $(A) \leftarrow result_{7-0}$ $(B) \leftarrow result_{15-8}$	The program counter is added by one unit, it multiplies the contents of the (A) and (B) registers, and it produces a result represented by 16 bits. The least significant 8 bits of the result are stored in the content of the accumulator (A) register, and the most significant 8 bits are stored in the content of the (B) register
--------	---	---	-----------	------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Division [all flags are affected, except the auxiliary carry-bit (AC)]

DIV AB	1	4	1000 0100	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow \text{quotient of the } (A)/(B)$ $(B) \leftarrow \text{rest of the } (A)/(B)$	The program counter is added by one unit and divides the content of the (A) register by the content of the (B) register. The division quotient is stored in the content of the accumulator (A) register, and the remainder is stored in the content of the (B) register
--------	---	---	-----------	--------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Decimal adjustment (conversion from binary/hexadecimal to BCD code) [carry-bit (C), zero (Z), and parity (P) flags are affected]					
DA A	1	1	1101 0100	$(PC) \leftarrow (PC) + 1$ $1^{\circ}) \text{ IF } \{(A_{3-0}) > 9\}$ $\text{OR } (AC) = 1\} \text{ then}$ $(A_{3-0}) \leftarrow (A_{3-0}) + 6;$ $2^{\circ}) \text{ IF } \{(A_{7-4}) > 9\}$ $\text{OR } (C) = 1\} \text{ then}$ $(A_{7-4}) \leftarrow (A_{7-4}) + 6;$	The program counter is added by one unit and adds the value 6 to the least significant 4 bits if $(AC) = 1$ or the inferior nibble does not belong to the decimal digits. It adds the value 6 to the most significant 4 bits if $(C) = 1$ or the superior nibble does not belong to the decimal digits

Logical operations [only the zero flag (Z) is affected and P if (A) is changed]

AND					
ANL A,Rn	1	1	0101 1rrr	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A) \text{ and } (Rn)$	The program counter is added by one unit, and the result of the logical AND operation between the contents of the accumulator (A) register and Rn register is stored in the content of the accumulator (A) register
ANL A, direct	2	1	0101 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow (A) \text{ and } (\text{direct})$	The program counter is added by two units, and the result of the logical AND operation between the contents of the accumulator (A) register and the position of internal RAM whose address is direct is stored in the content of the accumulator (A) register
ANL A, @Ri	1	1	0101 011I	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A) \text{ and } ((Ri))$	The program counter is added by one unit, and the result of the logical AND operation between the contents of the accumulator and position of the internal RAM whose address is

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
					given by the content of the Ri register is stored in the content of the accumulator (A) register
ANL A, #data	2	1	0101 0100 immediate data	(PC) \leftarrow (PC) + 2 (A) \leftarrow (A) and #data	The program counter is added by two units, and the result of the logical AND operation between the contents of the accumulator (A) register and the date value is stored in the content of the accumulator (A) register
ANL direct, A	2	1	0101 0010 direct address	(PC) \leftarrow (PC) + 2 (direc) \leftarrow (direc) and (A)	The program counter is added by two units, and the result of the logical AND operation between the contents of the internal RAM location, whose address is direct, and the accumulator (A) register is stored in the content of the internal RAM location, whose address is direct
ANL direct, #data	3	2	0101 0011 direct address immed. data	(PC) \leftarrow (PC) + 3 (direc) \leftarrow (direc) and #data	The program counter is added by three units, and the result of the logical AND operation between the contents of the position of internal RAM, whose address is direct, and the date value is stored in the content of the internal RAM memory location, whose address is direct

OR

ORL A,Rn	1	1	0100 1rrr	(PC) \leftarrow (PC) + 1 (A) \leftarrow (A) or (Rn)	The program counter is added by one unit, and the result of the logical operation OR between the contents of the accumulator
----------	---	---	-----------	------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
					(A) register and the Rn register is stored in the content of the accumulator (A) register
ORL A, direct	2	1	0100 0101 direct address	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow (A) \text{ or } (\text{direct})$	The program counter is added by two units, and the result of the logical operation OR between the contents of the accumulator and the position of internal RAM, whose address is direct, is stored in the content of the accumulator (A) register
ORL A, @Ri	1	1	0100 0111	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow (A) \text{ or } ((Ri))$	The program counter is added by one unit, and the result of the logical operation OR between the contents of the accumulator (A) register and the position of the internal RAM, whose address is given by the content of the Ri register, is stored in the content of the accumulator (A) register
ORL A, #data	2	1	0100 0100 immediate data	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow (A) \text{ or } \#data$	The program counter is added by two units, and the result of the logical operation OR between the contents of the accumulator (A) register and the date value is stored in the content of the accumulator (A) register
ORL direct, A	2	1	0100 0010 direct address	$(PC) \leftarrow (PC) + 2$ $(direc) \leftarrow (direc) \text{ or } (A)$	The program counter is added by two units, and the result of the logical operation OR between the contents of the internal RAM location, whose address is direct, and

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
ORL direct, #data	3	2	0100 0011 direct address immed. data	(PC) \leftarrow (PC) + 3 (direc) \leftarrow (direc) or #data	the accumulator (A) register is stored in the content of the internal RAM location, whose address is direct

Exclusive-OR

XRL A,Rn	1	1	0110 1rrr	(PC) \leftarrow (PC) + 1 (A) \leftarrow (A) or-ex (Rn)	The program counter is added by one unit, and the result of the OR-EX logic operation between the contents of the accumulator (A) register and the Rn register is stored in the content of the accumulator (A) register
XRL A, direct	2	1	0110 0101 direct address	(PC) \leftarrow (PC) + 2 (A) \leftarrow (A) or-ex (direct)	The program counter is added by two units, and the result of the OR-EX logical operation between the contents of the accumulator (A) register and the position of internal RAM, whose address is direct, is stored in the content of the accumulator (A) register
XRL A, @Ri	1	1	0110 011i	(PC) \leftarrow (PC) + 1 (A) \leftarrow (A) or-ex ((Ri))	The program counter is added by one unit, and the result of the OR-EX logical operation between the contents of the

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
					accumulator (A) register and the position of the internal RAM, whose address is given by the content of the Ri register, is stored in the content of the accumulator (A) register
XRL A, #data	2	1	0110 0100 immediate data	$(PC) \leftarrow (PC) + 2$ $(A) \leftarrow (A) \text{ or-ex } #data$	The program counter is added by two units, and the result of the OR-EX logical operation between the contents of the accumulator (A) register and the date value is stored in the content of the accumulator (A) register
XRL direct, A	2	1	0110 0010 direct address	$(PC) \leftarrow (PC) + 2$ $(direc) \leftarrow (direc)$ or-ex (A)	The program counter is added by two units, and the result of the OR-EX logical operation between the contents of the internal RAM location, whose address is direct, and the accumulator (A) register is stored in the content of the internal RAM location, whose address is direct
XRL direct, #data	3	2	0110 0011 direct address immed. data	$(PC) \leftarrow (PC) + 3$ $(direc) \leftarrow (direc)$ or-ex #data	The PC is added by three units, and the result of the OR logical operation between the contents of the internal RAM location, whose address is direct, and the date value is stored in the content off internal RAM position, whose address is direct

Reset [zero flag (Z) only]

CLR A	1	1	1110 0100	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow 00 \text{ h}$	The program counter is added by one unit, and the content of the accumulator (A) register is reset
-------	---	---	-----------	-------------------------------------------------------------	----------------------------------------------------------------------------------------------------

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Complementary (all flags are affected)					
CPL A	1	1	1111 0100	$(PC) \leftarrow (PC) + 1$ $(A) \leftarrow \text{not } (A)$	The program counter is added by one unit, and the content of the accumulator (A) register is stored with its complement of one
Rotation [the carry-bit (C) and zero (Z) flags are affected]					
RL A	1	1	0010 0011	$(PC) \leftarrow (PC) + 1$ $(An + 1) \leftarrow (An)$ for $n = 0$ to 6 $(A_0) = (C) \leftarrow (A_7)$	The program counter is added by one unit, and the content of the accumulator (A) register is rotated 1 bit to the left $(An + 1) \leftarrow (An)$ for $n = 0$ to 6 and $(A_0) = (C) \leftarrow (A_7)$, where n is the bit index. Ex: An for $n = 0$ corresponds to the 0 bit of the accumulator (A_0) , An for $n = 1$ corresponds to the 1 bit of the accumulator (A_1) , and so on
RLC A	1	1	0011 0011	$(PC) \leftarrow (PC) + 1$ $(An + 1) \leftarrow (An)$ for $n = 0$ to 6 $(A_0) \leftarrow (C) \text{ e } (C) \leftarrow (A_7)$	The program counter is added by one unit, and the content of the accumulator (A) register is rotated 1 bit to the left $(An + 1) \leftarrow (An)$ for $n = 0$ to 6 and $(C) \leftarrow (A_7)$ and $(A_0) \leftarrow (C)$, where n is the bit index
RR A	1	1	0000 0011	$(PC) \leftarrow (PC) + 1$ $(An) \leftarrow (An + 1)$ for $n = 0$ to 6 $(A_7) = (C) \leftarrow (A_0)$	The program counter is added by one unit, and the content of the accumulator (A) register is rotated from 1 bit to the right $(An) \leftarrow (An + 1)$ for $n = 0$ to 6 $(A_0) = (C) \leftarrow (A_7)$, where n is the bit index

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
RRC A	1	1	0001 0011	$(PC) \leftarrow (PC) + 1$ $(An) \leftarrow (An + 1)$ for n = 0 to 6 $(A_7) \leftarrow (C)$ e $(C) \leftarrow (A_0)$	The program counter is added by one unit, and the content of the accumulator (A) register is rotated from 1 bit to the right $(An) \leftarrow (An + 1)$ for n = 0 to 6 and $(C) \leftarrow (A_0)$ and $(A_7) \leftarrow (C)$, where n is the bit index

Change of the least significant 4 bits with the most significant 4 bits [zero flag (Z) only and parity flag (P)]

SWAP A	1	1	1100 0100	$(PC) \leftarrow (PC) + 1$ $(A_{3-0}) \leftrightarrow (A_{7-4})$	The program counter is added by one unit, and the content of the least significant 4 bits ($A_{3,0}$) is exchanged with the contents of the most significant 4 bits ($A_{7,4}$) of the content of the accumulator (A) register
--------	---	---	-----------	---------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Manipulation of Boolean variables

Reset (clear)

CLR C	1	1	1100 0011	$(PC) \leftarrow (PC) + 1$ $(C) \leftarrow \#0_2$	The program counter is added by one unit, and the carry-bit content is reset [only the carry-bit flag (C)]
CLR bit	2	1	1100 0010 bit address	$(PC) \leftarrow (PC) + 2$ $(bit) \leftarrow \#0_2$	The program counter is added by two units, and the content of the bit is reset (no flag is affected)

Set

SETB C	1	1	1101 0011	$(PC) \leftarrow (PC) + 1$ $(C) \leftarrow \#1_2$	The program counter is added by one unit, and the content of the carry-bit (C) flag is stored with the unit value [only the carry-bit (C) flag]
SETB bit	2	1	1101 0011 bit address	$(PC) \leftarrow (PC) + 2$ $(bit) \leftarrow \#1_2$	The program counter is added by two units, and the bit content is stored with the unit value (no flag is affected)

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Complement					
CPL C	1	1	1011 0011	$(PC) \leftarrow (PC) + 1$ $(C) \leftarrow \text{not } (C)$	The program counter is added by one unit, and the content of the carry-bit flag (C) is stored with its complement of one [carry-bit (C) flag only]
CPL bit	2	1	1011 0010	$(PC) \leftarrow (PC) + 2$ $(\text{bit}) \leftarrow \text{not } (\text{bit})$	The program counter is added by two units, and the content of the bit is stored with its complement of one (no flag is affected)
AND					
ANL C,bit	2	2	1000 0010 bit address	$(PC) \leftarrow (PC) + 2$ $(C) \leftarrow (C) \text{ and } (\text{bit})$	The program counter is added by two units, and the AND operation result between the carry-bit flag and bit contents is stored in the content of the carry-bit flag (C) [only the carry-bit (C) flag]
ANL C,/bit	2	2	1000 0000 bit address	$(PC) \leftarrow (PC) + 2$ $(C) \leftarrow (C) \text{ and } (\text{not } (\text{bit}))$	The program counter is added by two units, and the result of the AND operation between the contents of the carry-bit flag (C) and the complement of one of the bit is stored in the content of the carry-bit flag (C) (no flag is affected)
OR					
ORL C,bit	2	2	0111 0010 bit address	$(PC) \leftarrow (PC) + 2$ $(C) \leftarrow (C) \text{ or } (\text{bit})$	The program counter is added by two units, and the result of the OR operation between the contents of the carry-bit flag (C) and the bit is stored in the content of the carry-bit flag (C) [only the carry-bit (C) flag]

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
ORL C,/bit	2	2	1010 0000 bit address	$(PC) \leftarrow (PC) + 2$ $(C) \leftarrow (C) \text{ or } (\text{not}(\text{bit}))$	The program counter is added by two units, and the result of the OR operation between the contents of the carry-bit flag (C) and the complement of one of the bits is stored in the content of the carry-bit flag (C) (no flag is affected)

Move

MOV C,bit	2	1	1010 0010 bit address	$(PC) \leftarrow (PC) + 2$ $(C) \leftarrow (\text{bit})$	The program counter is added by two units, and the content of the bit is copied to the content of the carry-bit flag (C) [only the carry-bit (C) flag]
MOV bit, C	2	2	1001 0010 bit address	$(PC) \leftarrow (PC) + 2$ $(\text{bit}) \leftarrow (C)$	The program counter is added by two units, and the content of the carry-bit flag (C) is copied to the content of the bit (no flag is affected)

Conditional jump (no flag is affected)

JC address	2	2	0100 0000 rel address	$(PC) \leftarrow (PC) + 2$ If (C) = 1, then $(PC) \leftarrow \text{address}$	The program counter is added by two units, and if (C) = 1 then (PC) ← address (it jumps to the address of the program memory defined by the address); otherwise, the program executes the next instruction
JNC address	2	2	0101 0000 rel address	$(PC) \leftarrow (PC) + 2$ If (C) = 0, then $(PC) \leftarrow \text{address}$	The program counter is added by two units, and if (C) = 0, then (PC) ← address (it jumps to the address of the program memory defined by the address); otherwise, the program executes the next instruction

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
JB bit, address	3	2	0010 0000 bit address rel address	(PC) \leftarrow (PC) + 3 If (bit) = 1, then (PC) \leftarrow address	The program counter is added by three units, and if (bit) = 1, then (PC) \leftarrow address (it jumps to the address of the program memory defined by the address); otherwise, the program executes the next instruction
JNB bit, address	3	2	0011 0000 bit address rel address	(PC) \leftarrow (PC) + 3 If (bit) = 0, then (PC) \leftarrow address	The program counter is added by three units, and if (bit) = 0, then (PC) \leftarrow address (it jumps to the address of the program memory defined by the address); otherwise, the program executes the next instruction
JBC bit, address	3	2	0001 0000 bit address rel address	(PC) \leftarrow (PC) + 3 If (bit) = 1, then (bit) \leftarrow 0 (PC) \leftarrow address	The program counter is added by three units, and if (bit) = 1, then (PC) \leftarrow address (it jumps to the address of the program memory defined by the address) and zeroes the bit; otherwise, the program executes the next instruction
JZ address	2	2	0110 0000 rel address	(PC) \leftarrow (PC) + 2 If (A) = 0, then (PC) \leftarrow address	The program counter is added by two units, and if (A) = 0 [zero flag (Z) = 1], then (PC) \leftarrow address (it jumps to the address of the program memory defined by the address); otherwise, the program executes the next instruction
JNZ address	2	2	0111 0000 rel address	(PC) \leftarrow (PC) + 2 If (A) \neq 0, then (PC) \leftarrow address	The program counter is added by two units, and if (A) \neq 0 [zero flag (Z) = 0], then (PC) \leftarrow address (it jumps to the address of the program memory defined by the address); otherwise, the program executes the next instruction

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
Call the subroutine (no flag is affected)					
ACALL addr ₁₁	2	2	a ₁₀ a ₉ a ₈ 1 0001 a ₇ a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀	(PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP)) ← (PC ₇₋₀) (SP ← (SP) + 1 ((SP)) ← (PC ₁₅₋₈) (PC) ← addr ₁₁	It calls the subroutine located at the program memory address: The program counter is added by two units (it calculates the return address); it increments the content of the SP register (it points to the next stack address) by one unit; it stores the least significant byte of the content of the PC register in the content of the internal memory (stack), whose address is given by the content of the SP register; it increments the content of the SP register (it points to the next stack address) by one unit again; and it stores the most significant byte of the content of the PC register in the content of the position of internal RAM (stack), whose address is given by the content of the SP register
LCALL addr ₁₆	3	2	0001 0010 addr ₁₅₋₈ addr ₇₋₀	(PC) ← (PC) + 3 (SP) ← (SP) + 1 ((SP)) ← (PC ₇₋₀) (SP ← (SP) + 1 ((SP)) ← (PC ₁₅₋₈) (PC) ← addr ₁₆	It calls the subroutine located at the program memory address: The program counter is added by three units (it calculates return address); it increments the contents of the SP register (it points to the next stack address) by one unit; it stores the least significant byte of the content of the PC register in the content of the internal memory (stack), whose address is given by the contents of the SP register; it

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
					increments the content of the SP register (it points to the next stack address) by one unit; and it stores the most significant byte of the content of the PC register in the content of the position of internal RAM (stack), whose address is given by the contents of the SP register

Return of subroutines (no flag is affected)

RET	1	2	0010 0010	$(PC) \leftarrow (PC) + 1$ $(PC_{15-8}) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$ $(PC_{7-0}) \leftarrow ((SP))$ $(SP \leftarrow (SP) - 1$	It returns from the subroutine: it reads the address from the content of the stack and stores it in the content of the PC; the program counter is added by a unit; the content of the internal RAM (stack) position, whose address is given by the contents of the SP register, is copied to the content of the most significant byte of the program counter (PC) register; the content of the SP register (it points to the previous address of the stack) is decreased by one unit; the content of the internal RAM (stack) position, whose address is given by the contents of the SP register, is copied to the content of the least significant byte of the program counter (PC) register; and it decrements the content of the SP register by one unit (it points to the previous address of the stack)
-----	---	---	-----------	------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
RETI	1	2	0011 0010	$(PC) \leftarrow (PC) + 1$ $(PC_{15-8}) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$ $(PC_{7-0}) \leftarrow ((SP))$ $(SP \leftarrow (SP) - 1)$	<p>It returns from the service subroutine of an interruption source: it reads an address from the content of the stack and stores it in the contents of the PC; the program counter is added by a unit; the content of the internal RAM (stack) position, whose address is given by the contents of the SP register, is copied to the content of the most significant byte of the program counter (PC) register; it decreases by one unit the content of the SP register (it points to the previous address of the stack); the content of the internal RAM (stack) position, whose address is given by the content of the SP register, is copied to the content of the least significant byte of the program counter (PC) register; and it decrements the contents of the SP register by one unit (it points to the previous address of the stack)</p>

Unconditional jump (no flag is affected)

AJMP addr ₁₁	2	2	a ₁₀ a ₉ a ₈ 0 0001 a ₇ ... a ₀	(PC) $\leftarrow (PC) + 2$ $(PC_{10-0}) \leftarrow \text{addr}_{11}$	The program counter is added by two units, and the value addr ₁₁ is copied to the PC register
LJMP addr ₁₆	3	2	0000 0010 addr ₁₅₋₀ addr ₇₋₀	(PC) $\leftarrow (PC) + 3$ $(PC) \leftarrow \text{addr}_{15-0}$	The program counter is added by three units, and the value addr ₁₁ is copied to the PC register

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
SJMP address	2	2	1000 000 rel address	(PC) \leftarrow (PC) + 2 (PC) \leftarrow address	The program counter is added by two units, and the address value is copied to the PC register
JMP @A + DPTR	1	2	0111 0011	(PC) \leftarrow (PC) + 2 (PC) \leftarrow ((A) + (DPTR))	The program counter is added by two units, and the content of the internal RAM position, whose address is given by the sum of the contents of the A and DPTR registers, is copied to the PC register

Conditioned subroutine call [the carry-bit (C) and zero (Z) flags are affected]

CJNE A, direct, address	3	2	1011 0101 direct address rel address	(PC) \leftarrow (PC) + 3 If (A) \neq (direct), then it calls the subroutine whose address is direct If (A) < (direct), then (C) \leftarrow #1 ₂ ; otherwise, (C) \leftarrow #0 ₂	The program counter is added by three units, and if (A) \neq (direct), then it calls the subroutine whose address is [(PC) \leftarrow address], and it stores the return address of the subroutine in the stack (memory address of the instruction program following the instruction “CJNE A, direct, address,” and it performs the operations described in the LCALL address instruction); otherwise, the program executes the next instruction subsequent to that instruction. In addition, if (A) is less than the content of the internal RAM location whose address is direct, the carry-bit flag (C) is set; otherwise, it is reset
CJNE A, #data, address	3	2	1011 0100 immed. address rel address	(PC) \leftarrow (PC) + 3 If (A) \neq #data, then it calls the subroutine whose address is address	The program counter is added by three units, and if (A) \neq data value, then it calls the subroutine whose

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
				If $(A) < \#data$, then $(C) \leftarrow 1$; otherwise, $(C) \leftarrow 0$	address is $[(PC) \leftarrow address]$, and it stores the return address of the subroutine in the stack (address program memory of the instruction following the instruction “CJNE A, $\#data$, address,” and it performs the operations described in the LCALL address instruction); otherwise, the program executes the next instruction subsequent to that instruction. In addition, if (A) is less than data value, the carry-bit flag (C) is set; otherwise it is reset
CJNE Rn, $\#data$, address	3	2	1011 1rrr immed. address rel address	$(PC) \leftarrow (PC) + 3$ If $(Rn) \neq \#data$, then it calls the subroutine whose address is address If $(Rn) < \#data$, then $(C) \leftarrow 1$; otherwise, $(C) \leftarrow 0$	The program counter is added by three units, and if $(Rn) \neq \#data$ value, it calls the subroutine whose address is $[(PC) \leftarrow address]$, and it stores the return address of the subroutine in the stack (address program memory of the instruction following the instruction “CJNE Rn, $\#data$, address,” and it performs the operations described in the LCALL address instruction); otherwise, the program executes the next instruction subsequent to that instruction. In addition, if (Rn) is less than the data value, the carry-bit flag (C) is set; otherwise, it is reset

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
CJNE @Ri, #data, address	3	2	1011 011i immed. address rel address	(PC) \leftarrow (PC) + 3 If ((Ri)) \neq #data, then it calls the subroutine whose address is address If ((Ri)) < #data, then (C) \leftarrow 1; otherwise, (C) \leftarrow 0	The program counter is added with three units, and if ((Rn)) \neq date value, then it calls the subroutine whose address is address [(PC) \leftarrow address], and it stores the return address of the subroutine in the stack (address program memory of the instruction following the instruction “CJNE @Ri,#data, address,” and it performs the operations described in the LCALL address instruction); otherwise, the program executes the next instruction subsequent to that instruction. In addition, if ((Rn)) is less than data value, the carry-bit flag (C) is set; otherwise, it is reset

Deccrements a unit and jumps if nonzero [only zero flag (Z) is affected]

DJNZ Rn, address	2	2	1101 1rrr rel address	(PC) \leftarrow (PC) + 2 (Rn) \leftarrow (Rn)-1 IF (Rn) \neq 0 then (PC) \leftarrow address	The program counter is added by two units, the content of the Rn register is decremented by one unit, and if the content of the Rn register is different from zero, the program jumps to the address [(PC) \leftarrow address]
---------------------	---	---	--------------------------	--------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(continued)

Data transfer operations

Instruction	Byte	Cycles	Code	Symbolic representation	
DJNZ direct, adress	3	2	1101 010 direct address rel address	$(PC) \leftarrow (PC) + 3$ $(direct) \leftarrow (direct) - 1$ If $(direct) \neq 0$, then $(PC) \leftarrow address$	The program counter is added by three units; the content of the position of internal RAM, whose address is given by direct, is decreased by one unit; and if the content of the position of the internal RAM, whose address is given by direct, is different from zero, the program jumps to the address $[(PC) \leftarrow address]$

Não executa nada (nenhum flag é afetado)

NOP	1	1	0000 0000	$(PC) \leftarrow (PC) + 1$	O contador de programa é somado de uma unidade
-----	---	---	-----------	----------------------------	------------------------------------------------

Index

A

Addresses of service subroutines, 235
Addressing by register, 102–103
Addressing modes, 74, 75, 102–106
Assembly programming, 131–165
Asynchronous, 272, 273
Auxiliary carry-bit flag (AC), 96–98, 100

B

Baud rate, 273–280, 283, 285, 286, 288, 290
Bounce, 207–213, 216–219, 221–223

C

Calling a subroutine, 167, 172–180
Carry-bit flag (C), 95–97, 100, 102, 109, 112, 113, 117, 120, 122, 125, 126
Clock, 65–69, 75, 88–90
Combined or mixed addressing, 102, 105–106
Computer systems, 1–59

D

Data memory, 61, 64–67, 69, 72–89, 92, 93
Direct addressing, 103

E

External interruptions, 229–230, 233–236, 239

F

Firmware, 42, 44, 45, 57, 59
Flip-flop, 3, 11, 12, 19, 20, 43
Flowchart, 9, 10, 46–51, 53, 59, 131–165
Full duplex, 273

H

Hardware, 19, 24, 31, 33, 42, 44, 45, 49, 53–55, 57

I

Immediate addressing, 104
Indirect by base registers, 102, 104
Indirect or indexed by register, 102–106
Input port, 191–223
Instructions, 95–117, 119–121, 126, 127
Instruction set, 95–129
Internal architecture, 61–64
Internal Oscillator (Clock), 21–22
Interruption enable (IE), 226, 233, 235
Interruption priority, 233, 234
Interruption sources, 225–236, 239

M

Machine cycle, 68, 69, 87–89
MCS-51, 61, 62, 64, 70, 71, 74, 78, 88, 89, 92, 93
Memory, 1, 11–21, 26, 34, 35, 41–51, 53, 55–58
Memory organization, 64–80
Microcomputer, 21, 41–45, 57, 58
Microcontroller, 21–23, 29–34, 37, 38, 45–46, 48, 54–57, 59
Microprocessor, 1, 21, 22, 41–49, 51, 54–59
Mode 0, 242, 245, 247–249, 251, 267, 275–276, 279–281, 283
Mode 1, 243, 245, 249, 251, 258, 267, 268, 274–280, 283, 285, 286, 288, 290
Mode 2, 243, 246, 249, 258, 267, 268, 276–280, 285, 288, 290
Mode 3, 244, 246, 250–251, 278–280, 290

N

Numeral systems, 1–7, 58

O

Operation modes, 242–245, 247–251

Output port, 191–223

Overflow-bit flag (OV), 98–99

P

Parity bit, 273–275, 277, 278

Pin description, 64

Program memory, 61, 64–73, 80–82, 88, 89, 92, 93

Programming, 13, 15, 17, 18, 42, 44, 47–54, 59

Program status word (PSW), 95–102, 114, 117–128

Pulse width modulation (PWM), 36–38

Q

Queue, 167–168

R

Register, 3, 4, 7, 19, 41–44, 54–58

Registers' banks, 74–76

Reset, 22–23, 57, 58

Reset (initialization) signal, 68, 71, 72, 75, 87–88, 90, 91, 93, 94

Returning of a subroutine, 167, 172–180

S

SBUF, *see* Serial buffer (SBUF)

SCON, *see* Serial control (SCON)

Sequential software, 132, 136–139, 144

Serial buffer (SBUF), 274–278, 280, 281, 283, 285, 286, 289

Serial communication interface, 271–319

Serial control (SCON), 274–281, 283, 285, 288

SFRs, *see* Special function registers (SFRs)

Software with loop, 140–143

Special function registers (SFRs), 65, 67, 68, 73, 74, 78–80, 87, 88, 91, 92, 94

Stack, 167–180, 182, 187

Stack pointer (SP), 168–180

Start bit, 273, 274, 276–278, 280

Stop bit, 273–279

Subroutine, 167–188

T

Timer control (TCON), 246

Timer mode (TMOD), 245, 250, 251

Timers/counters, 226, 229–235, 241–269

Timing routines, 204–206

TMOD, *see* Timer mode (TMOD)

V

Variable management, 225–229

Z

Zero-bit flag (Z), 101–102