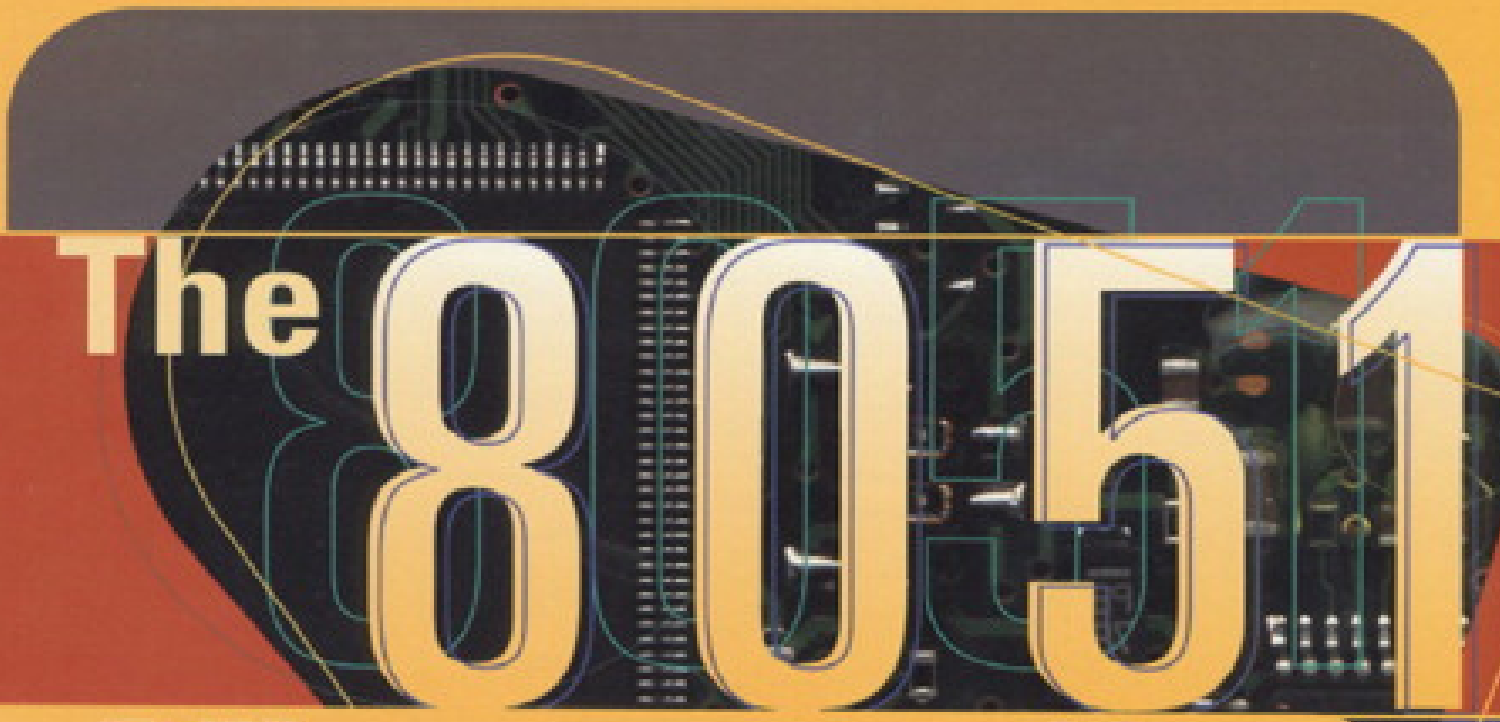


The 8051 Microcontroller and Embedded Systems

Muhammad Ali Mazidi
Janice Gillispie Mazidi

THE 8051 MICROCONTROLLER AND EMBEDDED SYSTEMS

Muhammad Ali Mazidi
Janice Gillispie Mazidi

Prentice Hall
Upper Saddle River, New Jersey *Columbus, Ohio*

.. man's glory lieth in his knowledge,
his upright conduct, his praiseworthy character,
his wisdom, and not in his nationality or rank.

Baha'u'llah

CONTENTS AT A GLANCE

CHAPTERS

APPENDICES

CONTENTS

INTRODUCTION

Products using microprocessors generally fall into two categories. The first category uses high-performance microprocessors such as the Pentium in applications where system performance is critical. We have an entire book dedicated to this topic, *The 80x86 IBM PC and Compatible Computers, Volumes I and II*, from Prentice Hall. In the second category of applications, performance is secondary; issues of space, power, and rapid development are more critical than raw processing power. The microprocessor for this category is often called a microcontroller.

This book is for the second category of applications. The 8051 is a widely used microcontroller. There are many reasons for this, including the existence of multiple producers and its simple architecture. This book is intended for use in college-level courses teaching microcontrollers and embedded systems. It not only establishes a foundation of assembly language programming, but also provides a comprehensive treatment of 8051 interfacing for engineering students. From this background, the design and interfacing of microcontroller-based embedded systems can be explored. This book can be also used by practicing technicians, hardware engineers, computer scientists, and hobbyists. It is an ideal source for those building stand-alone projects, or projects in which data is collected and fed into a PC for distribution on a network.

Prerequisites

Readers should have had an introductory digital course. Knowledge of a programming language would be helpful but is not necessary. Although the book is written for those with no background in assembly language programming, students with prior assembly language experience will be able to gain a mastery of 8051 architecture very rapidly and start on their projects right away.

Overview

A systematic, step-by-step approach is used to cover various aspects of 8051 Assembly language programming and interfacing. Many examples and sample programs are given to clarify the concepts and provide students with an opportunity to learn by doing. Review questions are provided at the end of each section to reinforce the main points of the section.

Chapter 0 covers number systems (binary, decimal, and hex), and provides an introduction to basic logic gates and computer terminology. This is designed especially for students, such as mechanical engineering students, who have not taken a digital logic course or those who need to refresh their memory on these topics.

Chapter 1 discusses 8051 history and features of other 8051 family members such as the 8751, 89C51, DS5000, and 8031. It also provides a list of various producers of 8051 chips.

Chapter 2 discusses the internal architecture of the 8051 and explains the use of an 8051 assembler to create ready-to-run programs. It also explores the stack and the flag register.

In Chapter 3 the topics of loop, jump, and call instructions are discussed, with many programming examples.

Chapter 4 is dedicated to the discussion of I/O ports. This allows students who are working on a project to start experimenting with 8051 I/O interfacing and start the project as soon as possible.

Chapter 5 covers the 8051 addressing modes and explains how to use the code space of the 8051 to store data, as well as how to access data.

Chapter 6 is dedicated to arithmetic instructions and programs.

Logic instructions and programs are covered in Chapter 7.

In Chapter 8 we discuss one of the most important features of the 8051, bit manipulation, as well as single-bit instructions of the 8051.

Chapter 9 describes the 8051 timers and how to use them as event-counters.

Chapter 10 is dedicated to serial data communication of the 8051 and its interfacing to the RS232. It also shows 8051 communication with COM ports of the IBM PC and compatible computers.

Chapter 11 provides a detailed discussion of 8051 interrupts with many examples on how to write interrupt handler programs.

Chapter 12 shows 8051 interfacing with real-world devices such as LCDs, ADCs, and sensors.

Chapter 13 shows 8051 interfacing with real world devices such as the keyboard, stepper motors, and DAC devices.

In Chapter 14 we cover 8031/51 interfacing with external memories, both ROM and RAM.

Finally, in Chapter 15 the issue of adding more ports to the 8031/51 is discussed, and the interfacing of an 8255 chip with the microcontroller is covered in detail.

The appendices have been designed to provide all reference material required for the topics covered in the book. Appendix A describes each 8051 instruction in detail, with examples. Appendix A also provides the clock count for instructions, 8051 register diagrams, and RAM memory maps. Appendix B describes wire wrapping, and how to design your own 8051 trainer board based on 89C51 or DS5000 chips. Appendix C covers IC technology and logic families, as well as 8051 I/O port interfacing and fan-out. Make sure you study this before connecting the 8051 to an external device. In Appendix D, the use of flowcharts and psuedocode is explored. Appendix E is for students familiar with x86 architecture who need to make a rapid transition to 8051 architecture. Appendix F provides the table for ASCII characters. Appendix G lists resources for assembler shareware, and electronics parts. Appendix H contains data sheets for the 8051 and other IC chips.

Diskette contents

The diskette attached to the book contains the lab manual, which has many experiments for software programming and hardware interfacing of the 8051. These are in Microsoft Word 97 format. In addition, the diskette contains the source code for all the programs in the book (in ASCII files). Also on the diskette are two guides for using 8051 assemblers and simulators from Franklin Software and Keil Corporation.

Acknowledgments

This book is the result of the dedication and encouragement of many individuals. Our sincere and heartfelt appreciation goes to all of them.

First, we would like to thank Professor Danny Morse, the most knowledgeable and experienced person on the 8051 that we know. He felt a strong need for a book such as this, and due to his lack of time he encouraged us to write it. He is the one who introduced us to this microcontroller and was always there, ready to discuss issues related to 8051 architecture.

Also we would like to express our sincere thanks to Professor Clyde Knight of Devry Institute of Technology for his helpful suggestions on the organization of the book.

In addition, the following professors and students found errors while using the book in its pre-publication form in their microcontroller course, and we thank them sincerely: Professor Phil Golden and John Berry of DeVry Institute of Technology, Robert Wrightson, Priscilla Martinez, Benjamin Fombon, David Bergman, John Higgins, Scot Robinson, Jerry Chrane, James Piott, Daniel Rusert, Michael Beard, Landon Hull, Jose Lopez, Larry Hill, David Johnson, Jerry Kelso, Michael Marshall, Marc Hoang, Trevor Isra.

Mr. Rolin McKinlay, an excellent student of the 8051, made many valuable suggestions, found many errors, and helped to produce the solution manual for the end-of-chapter problems. We sincerely appreciate his enthusiasm for this book.

Finally, we would like to thank the people at Prentice Hall, in particular our publisher, Mr. Charles Stewart, who continues to support and encourage our writing, and our production editor Alex Wolf who made the book a reality.

We enjoyed writing this book, and hope you enjoy reading it and using it for your courses and projects. Please let us know if you have any suggestions or find any errors.

Assemblers

The following gives two sites where you can download assemblers:

www.fsinc.com	for Franklin Software, Inc.
www.keil.com	for Keil Corporation

Another interesting web site is www.8052.com for more discussion on the microcontroller. Finally, the following site provides useful Intel manuals:

http://developer.intel.com/design/auto/mcs51/manuals

ABOUT THE AUTHORS

Muhammad Ali Mazidi holds Master's degrees from both Southern Methodist University and the University of Texas at Dallas, and currently is completing his Ph.D. in the Electrical Engineering Department of Southern Methodist University. He is a co-founder and chief researcher of Microprocessor Education Group, a company dedicated to bringing knowledge of microprocessors to the widest possible audience. He also teaches microprocessor-based system design at DeVry Institute of Technology in Dallas, Texas.

Janice Gillispie Mazidi has a Master of Science degree in Computer Science from the University of North Texas. After several years experience as a software engineer in Dallas, she co-founded Microprocessor Education Group, where she is the chief technical writer and production manager, and is responsible for software development and testing.

The Mazidis have been married since 1985 and have two sons, Robert Nabil and Michael Jamal.

The authors can be contacted at the following address if you have any comments or suggestions, or if you find any errors.

Microprocessor Education Group
P.O. Box 381970
Duncanville, TX 75138
U.S.A.

mmazidi@dal.devry.edu

This volume is dedicated to the memory of Dr. A. Davoodi, Professor of Tehran University, who in the tumultuous years of my youth taught me the importance of an independent search for truth. -- Muhammad Ali Mazidi

CHAPTER 0

INTRODUCTION TO COMPUTING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **Convert any number from base 2, base 10, or base 16 to any of the other two bases**
- **Add and subtract hex numbers**
- **Add binary numbers**
- **Represent any binary number in 2's complement**
- **Represent an alphanumeric string in ASCII code**
- **Describe logical operations AND, OR, NOT, XOR, NAND, NOR**
- **Use logic gates to diagram simple circuits**
- **Explain the difference between a bit, a nibble, a byte, and a word**
- **Give precise mathematical definitions of the terms *kilobyte*, *megabyte*, *terabyte*, and *gigabyte***
- **Explain the difference between RAM and ROM and describe their use**
- **Describe the purpose of the major components of a computer system**
- **List the three types of buses found in computers and describe the purpose of each type of bus**
- **Describe the role of the CPU in computer systems**
- **List the major components of the CPU and describe the purpose of each**

To understand the software and hardware of a microcontroller-based system, one must first master some very basic concepts underlying computer design. In this chapter (which in the tradition of digital computers can be called Chapter 0), the fundamentals of numbering and coding systems are presented. After an introduction to logic gates, an overview of the workings inside the computer is given. Finally, in the last section we give a brief history of CPU architecture. Although some readers may have an adequate background in many of the topics of this chapter, it is recommended that the material be scanned, however briefly.

SECTION 0.1: NUMBERING AND CODING SYSTEMS

Whereas human beings use base 10 *(decimal)* arithmetic, computers use the base 2 (*binary*) system. In this section we explain how to convert from the decimal system to the binary system, and vice versa. The convenient representation of binary numbers, called *hexadecimal,* also is covered. Finally, the binary format of the alphanumeric code, called *ASCII*, is explored.

Decimal and binary number systems

Although there has been speculation that the origin of the base 10 system is the fact that human beings have 10 fingers, there is absolutely no speculation about the reason behind the use of the binary system in computers. The binary system is used in computers because 1 and 0 represent the two voltage levels of on and off. Whereas in base 10 there are 10 distinct symbols, 0, 1, 2, ..., 9, in base 2 there are only two, 0 and 1, with which to generate numbers. Base 10 contains digits 0 through 9; binary contains digits 0 and 1 only. These two binary digits, 0 and 1, are commonly referred to as *bits*.

Converting from decimal to binary

One method of converting from decimal to binary is to divide the decimal number by 2 repeatedly, keeping track of the remainders. This process continues until the quotient becomes zero. The remainders are then written in reverse order to obtain the binary number. This is demonstrated in Example 0-1.

Example 0-1

Convert 25_{10} to binary.

Solution:

	Quotient	*Remainder*	
25/2 =	12	1	LSB (least significant bit)
12/2 =	6	0	
6/2 =	3	0	
3/2 =	1	1	
1/2 =	0	1	MSB (most significant bit)

Therefore, $25_{10} = 11001_2$.

Converting from binary to decimal

740683_{10} =

3×10^0	=	3
8×10^1	=	80
6×10^2	=	600
0×10^3	=	0000
4×10^4	=	40000
7×10^5	=	700000
		740683

To convert from binary to decimal, it is important to understand the concept of weight associated with each digit position. First, as an analogy, recall the weight of numbers in the base 10 system, as shown in the diagram. By the same token, each digit position in a number in base 2 has a weight associated with it:

110101_2 =

			Decimal	*Binary*
1×2^0	=	1×1 =	1	1
0×2^1	=	0×2 =	0	00
1×2^2	=	1×4 =	4	100
0×2^3	=	0×8 =	0	0000
1×2^4	=	1×16 =	16	10000
1×2^5	=	1×32 =	32	100000
			53	110101

Knowing the weight of each bit in a binary number makes it simple to add them together to get its decimal equivalent, as shown in Example 0-2.

Example 0-2

Convert 11001_2 to decimal.

Solution:

Weight:	16	8	4	2	1
Digits:	1	1	0	0	1
Sum:	16 +	8 +	0 +	0 +	$1 = 25_{10}$

Knowing the weight associated with each binary bit position allows one to convert a decimal number to binary directly instead of going through the process of repeated division. This is shown in Example 0-3.

Example 0-3

Use the concept of weight to convert 39_{10} to binary.

Solution:

Weight:	32	16	8	4	2	1
	1	0	0	1	1	1
	32 +	0 +	0 +	4 +	2 +	$1 = 39$

Therefore, $39_{10} = 100111_2$.

Hexadecimal system

Base 16, the *hexadecimal* system as it is called in computer literature, is used as a convenient representation of binary numbers. For example, it is much easier for a human being to represent a string of 0s and 1s such as 100010010110 as its hexadecimal equivalent of 896H. The binary system has 2 digits, 0 and 1. The base 10 system has 10 digits, 0 through 9. The hexadecimal (base 16) system has 16 digits. In base 16, the first 10 digits, 0 to 9, are the same as in decimal, and for the remaining six digits, the letters A, B, C, D, E, and F are used. Table 0-1 shows the equivalent binary, decimal, and hexadecimal representations for 0 to 15.

Table 0-1: Base 16 Number Systems

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

Converting between binary and hex

To represent a binary number as its equivalent hexadecimal number, start from the right and group 4 bits at a time, replacing each 4-bit binary number with its hex equivalent shown in Table 0-1. To convert from hex to binary, each hex digit is replaced with its 4-bit binary equivalent. See Examples 0-4 and 0-5.

Example 0-4

Represent binary 100111110101 in hex.

Solution:
First the number is grouped into sets of 4 bits: 1001 1111 0101.
Then each group of 4 bits is replaced with its hex equivalent:

```
1001  1111  0101
 9     F     5
```

Therefore, 100111110101_2 = 9F5 hexadecimal.

Example 0-5

Convert hex 29B to binary.

Solution:

```
      2     9     B
=   0010  1001  1011
```

Dropping the leading zeros gives 1010011011.

Converting from decimal to hex

Converting from decimal to hex could be approached in two ways:

1. Convert to binary first and then convert to hex. Example 0-6 shows this method of converting decimal to hex.
2. Convert directly from decimal to hex by repeated division, keeping track of the remainders. Experimenting with this method is left to the reader.

Example 0-6

(a) Convert 45_{10} to hex.

32	16	8	4	2	1
1	0	1	1	0	1

First, convert to binary.
$32 + 8 + 4 + 1 = 45$

$45_{10} = 0010\ 1101_2 = 2D$ hex

(b) Convert 629_{10} to hex.

512	256	128	64	32	16	8	4	2	1
1	0	0	1	1	1	0	1	0	1

$629_{10} = (512 + 64 + 32 + 16 + 4 + 1) = 0010\ 0111\ 0101_2 = 275$ hex

(c) Convert 1714_{10} to hex.

1024	512	256	128	64	32	16	8	4	2	1
1	1	0	1	0	1	1	0	0	1	0

$1714_{10} = (1024 + 512 + 128 + 32 + 16 + 2) = 0110\ 1011\ 0010_2 = 6B2$ hex

Converting from hex to decimal

Conversion from hex to decimal can also be approached in two ways:

1. Convert from hex to binary and then to decimal. Example 0-7 demonstrates this method of converting from hex to decimal.
2. Convert directly from hex to decimal by summing the weight of all digits.

Example 0-7

Convert the following hexadecimal numbers to decimal.

(a) $6B2_{16} = 0110\ 1011\ 0010_2$

1024	512	256	128	64	32	16	8	4	2	1
1	1	0	1	0	1	1	0	0	1	0

$1024 + 512 + 128 + 32 + 16 + 2 = 1714_{10}$

(b) $9F2D_{16} = 1001\ 1111\ 0010\ 1101_2$

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	0	1	1	1	1	1	0	0	1	0	1	1	0	1

$32768 + 4096 + 2048 + 1024 + 512 + 256 + 32 + 8 + 4 + 1 = 40{,}749_{10}$

Table 0-2: Counting in Bases

Decimal	Binary	Hex
0	00000	0
1	00001	1
2	00010	2
3	00011	3
4	00100	4
5	00101	5
6	00110	6
7	00111	7
8	01000	8
9	01001	9
10	01010	A
11	01011	B
12	01100	C
13	01101	D
14	01110	E
15	01111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1A
27	11011	1B
28	11100	1C
29	11101	1D
30	11110	1E
31	11111	1F

Counting in bases 10, 2, and 16

To show the relationship between all three bases, in Table 0-2 we show the sequence of numbers from 0 to 31 in decimal, along with the equivalent binary and hex numbers. Notice in each base that when one more is added to the highest digit, that digit becomes zero and a 1 is carried to the next-highest digit position. For example, in decimal, 9 + 1 = 0 with a carry to the next-highest position. In binary, 1 + 1 = 0 with a carry; similarly, in hex, F + 1 = 0 with a carry.

Table 0 - 3: Binary Addition

A + B	Carry	Sum
0 + 0	0	0
0 + 1	0	1
1 + 0	0	1
1 + 1	1	0

Addition of binary and hex numbers

The addition of binary numbers is a very straightforward process. Table 0-3 shows the addition of two bits. The discussion of subtraction of binary numbers is bypassed since all computers use the addition process to implement subtraction. Although computers have adder circuitry, there is no separate circuitry for subtractors. Instead, adders are used in conjunction with *2's complement* circuitry to perform subtraction. In other words, to implement "$x - y$", the computer takes the 2's complement of y and adds it to x. The concept of 2's complement is reviewed next. Example 0-8 shows the addition of binary numbers.

Example 0-8

Add the following binary numbers. Check against their decimal equivalents.

Solution:

	Binary	*Decimal*
	1101	13
+	1001	9
	10110	22

2's complement

To get the 2's complement of a binary number, invert all the bits and then add 1 to the result. Inverting the bits is simply a matter of changing all 0s to 1s and 1s to 0s. This is called the *1's complement*. See Example 0-9.

Example 0-9

Take the 2's complement of 10011101.

Solution:

```
  10011101    binary number
  01100010    1's complement
+        1
  01100011    2's complement
```

Addition and subtraction of hex numbers

In studying issues related to software and hardware of computers, it is often necessary to add or subtract hex numbers. Mastery of these techniques is essential. Hex addition and subtraction are discussed separately below.

Addition of hex numbers

This section describes the process of adding hex numbers. Starting with the least significant digits, the digits are added together. If the result is less than 16, write that digit as the sum for that position. If it is greater than 16, subtract 16 from it to get the digit and carry 1 to the next digit. The best way to explain this is by example, as shown in Example 0-10.

Example 0-10

Perform hex addition: 23D9 + 94BE.

Solution:

```
  23D9      LSD:  9 + 14 = 23        23 - 16 = 7 with a carry
+ 94BE            1 + 13 + 11 = 25  25 - 16 = 9 with a carry
  B897            1 + 3 + 4 = 8
            MSD:  2 + 9 = B
```

Subtraction of hex numbers

In subtracting two hex numbers, if the second digit is greater than the first, borrow 16 from the preceding digit. See Example 0-11.

ASCII code

The discussion so far has revolved around the representation of number systems. Since all information in the computer must be represented by 0s and 1s, binary patterns must be assigned to letters and other characters. In the 1960s a standard representation called *ASCII* (American Standard Code for Information Interchange) was established. The ASCII (pronounced "ask-E") code assigns

binary patterns for numbers 0 to 9, all the letters of the English alphabet, both uppercase (capital) and lowercase, and many control codes and punctuation marks. The great advantage of this system is that it is used by most computers, so that information can be shared among computers. The ASCII system uses a total of 7 bits to represent each code. For example, 100 0001 is assigned to the uppercase letter "A" and 110 0001 is for the lowercase "a". Often, a zero is placed in the most significant bit position to make it an 8-bit code. Figure 0-1 shows selected ASCII codes. A complete list of ASCII codes is given in Appendix F. The use of ASCII is not only standard for keyboards used in the United States and many other countries but also provides a standard for printing and displaying characters by output devices such as printers and monitors.

Hex	*Symbol*	*Hex*	*Symbol*
41	A	61	a
42	B	62	b
43	C	63	c
44	D	64	d
...	...	...	...
59	Y	79	y
5A	Z	7A	z

Figure 0-1. Selected ASCII Codes

Notice that the pattern of ASCII codes was designed to allow for easy manipulation of ASCII data. For example, digits 0 through 9 are represented by ASCII codes 30 through 39. This enables a program to easily convert ASCII to decimal by masking off the "3" in the upper nibble. Also notice that there is a relationship between the uppercase and lowercase letters. The uppercase letters are represented by ASCII codes 41 through 5A while lowercase letters are represented by codes 61 through 7A. Looking at the binary code, the only bit that is different between the uppercase "A" and lowercase "a" is bit 5. Therefore, conversion between uppercase of lowercase is as simple as changing bit 5 of the ASCII code.

Example 0-11

Perform hex subtraction: 59F - 2B8.

Solution:

```
  59F        LSD:  8 from 15 = 7
- 2B8              11 from 25 (9 + 16) = 14 (E)
  2E7              2 from 4 (5 - 1) = 2
```

Review Questions

1. Why do computers use the binary number system instead of the decimal system?
2. Convert 34_{10} to binary and hex.
3. Convert 110101_2 to hex and decimal.
4. Perform binary addition: 101100 + 101.
5. Convert 101100_2 to its 2's complement representation.
6. Add 36BH + F6H.
7. Subtract 36BH - F6H.
8. Write "80x86 CPUs" in its ASCII code (in hex form).

SECTION 0.2: DIGITAL PRIMER

This section gives an overview of digital logic and design. First, we cover binary logic operations, then we show gates that perform these functions. Next, logic gates are put together to form simple digital circuits. Finally, we cover some logic devices commonly found in microcontroller interfacing.

Binary logic

As mentioned earlier, computers use the binary number system because the two voltage levels can be represented as the two digits 0 and 1. Signals in digital electronics have two distinct voltage levels. For example, a system may define 0 V as logic 0 and +5 V as logic 1. Figure 0-2 shows this system with the built-in tolerances for variations in the voltage. A valid digital signal in this example should be within either of the two shaded areas.

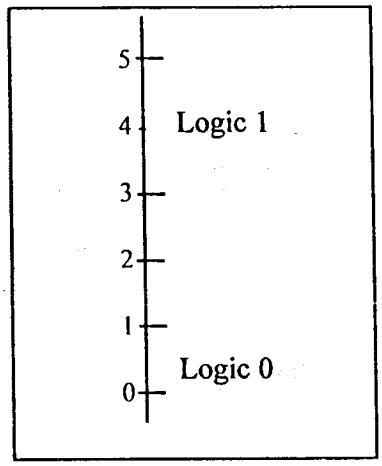


Figure 0-2. Binary Signals

Logic gates

Binary logic gates are simple circuits that take one or more input signals and send out one output signal. Several of these gates are defined below.

AND gate

The AND gate takes two or more inputs and performs a logic AND on them. See the truth table and diagram of the AND gate. Notice that if both inputs to the AND gate are 1, the output will be 1. Any other combination of inputs will give a 0 output. The example shows two inputs, x and y. Multiple outputs are also possible for logic gates. In the case of AND, if all inputs are 1, the output is 1. If any input is 0, the output is zero.

Logical AND Function

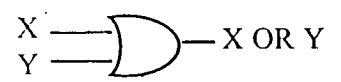
Inputs	Output
X Y	X AND Y
0 0	0
0 1	0
1 0	0
1 1	1

OR gate

The OR logic function will output a 1 if one or more inputs is 1. If all inputs are 0, then and only then will the output be 0.

Logical OR Function

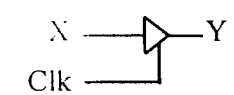
Inputs	Output
X Y	X OR Y
0 0	0
0 1	1
1 0	1
1 1	1



Tri-state buffer

A buffer gate does not change the logic level of the input. It is used to isolate or amplify the signal.

Buffer

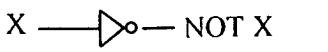


Inverter

The inverter, also called NOT, outputs the value opposite to that input to the gate. That is, a 1 input will give a 0 output, while a 0 input will give a 1 output.

Logical Inverter

Input	Output
X	NOT X
0	1
1	0

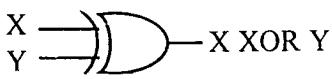


XOR gate

The XOR gate performs an exclusive-OR operation on the inputs. Exclusive-OR produces a 1 output if one (but only one) input is 1. If both operands are 0, the output is zero. Likewise, if both operands are 1, the output is also zero. Notice from the XOR truth table, that whenever the two inputs are the same, the output is zero. This function can be used to compare two bits to see if they are the same.

Logical XOR Function

Inputs	Output
X Y	X XOR Y
0 0	0
0 1	1
1 0	1
1 1	0

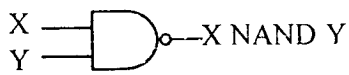


NAND and NOR gates

The NAND gate functions like an AND gate with an inverter on the output. It produces a zero output when all inputs are 1; otherwise, it produces a 1 output. The NOR gate functions like an OR gate with an inverter on the output. It produces a 1 if all inputs are 0; otherwise, it produces a 0. NAND and NOR gates are used extensively in digital design because they are easy and inexpensive to fabricate. Any circuit that can be designed with AND, OR, XOR, and INVERTER gates can be implemented using only NAND and NOR gates. A simple example of this is given below. Notice in NAND, that if any input is zero, the output is one. Notice in NOR, that if any input is one, the output is zero.

Logical NAND Function

Inputs	Output
X Y	X NAND Y
0 0	1
0 1	1
1 0	1
1 1	0

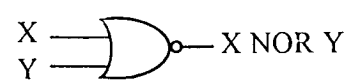


Logic design using gates

Next we will show a simple logic design to add two binary digits. If we add two binary digits there are four possible outcomes:

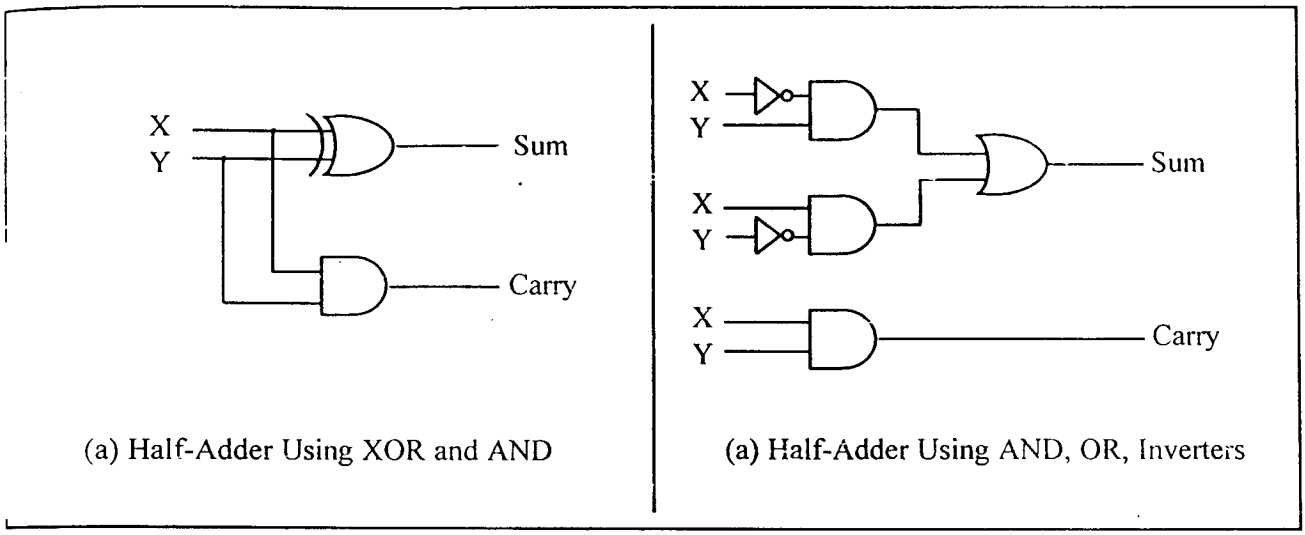
Logical NOR Function

Inputs	Output
X Y	X NOR Y
0 0	1
0 1	0
1 0	0
1 1	0



	Carry	*Sum*
0 + 0 =	0	0
0 + 1 =	0	1
1 + 0 =	0	1
1 + 1 =	1	0

Notice that when we add 1 + 1 we get 0 with a carry to the next higher place. We will need to determine the sum and the carry for this design. Notice that the sum column above matches the output for the XOR function, and that the carry column matches the output for the AND function. Figure 0-3 (a) shows a simple adder implemented with XOR and AND gates. Figure 0-3 (b) shows the same logic circuit implemented with AND and OR gates.



(a) Half-Adder Using XOR and AND

(a) Half-Adder Using AND, OR, Inverters

Figure 0-3. Two Implementations of a Half-Adder

Figure 0-4 shows a block diagram of a half-adder. Two half-adders can be combined to form an adder that can add three input digits. This is called a full-adder. Figure 0-5 shows the logic diagram of a full adder, along with a block diagram which masks the details of the circuit. Figure 0-6 shows a 3-bit adder using 3 full-adders.

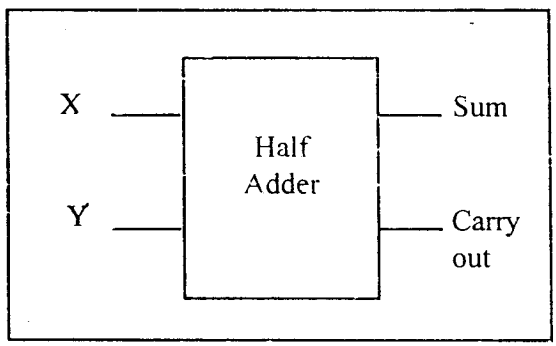


Figure 0-4. Block Diagram of a Half-Adder

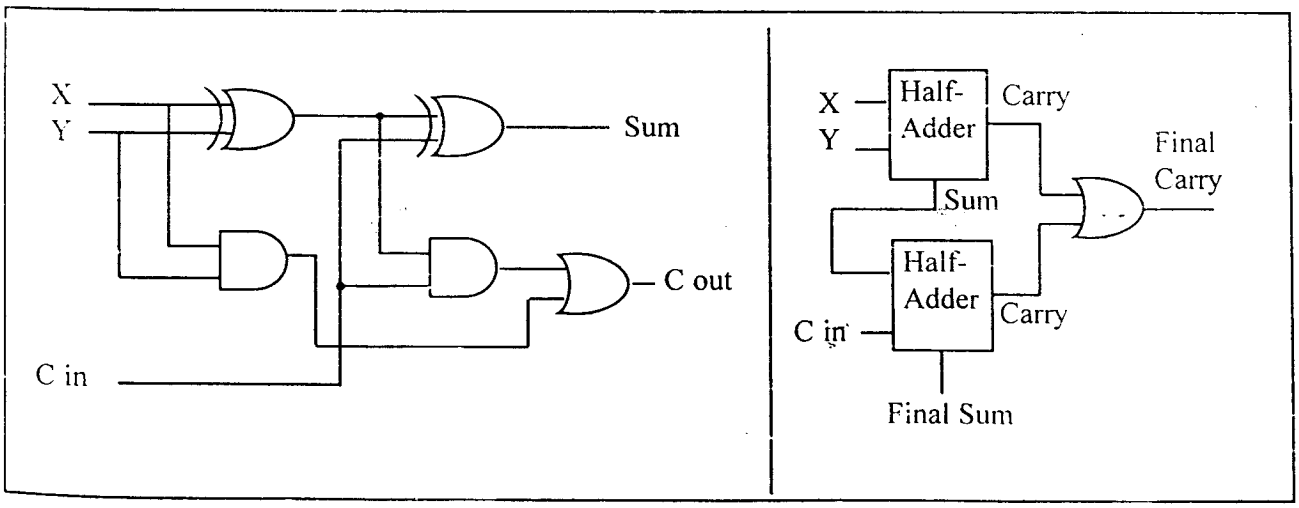


Figure 0-5. Full-Adder Built From a Half-Adder

Decoders

Another example of the application of logic gates is the decoder. Decoders are widely used for address decoding in computer design. Figure 0-7 shows decoders for 9 (1001 binary), and 5 (0101) using inverters and AND gates.

Flip-flops

A widely used component in digital systems is the flip-flop. Frequently, flip-flops are used to store data. Figure 0-8 shows the logic diagram, block diagram, and truth table for a flip-flop.

The D flip-flop is widely used to latch data. Notice from the truth table that a D-FF grabs the data at the input as the clock is activated. A D-FF holds the data as long as the power is on.

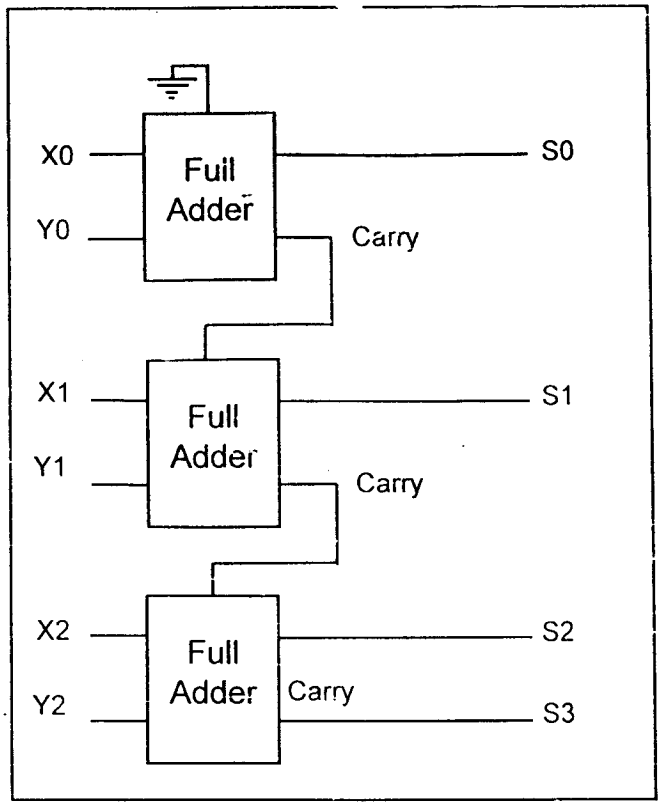
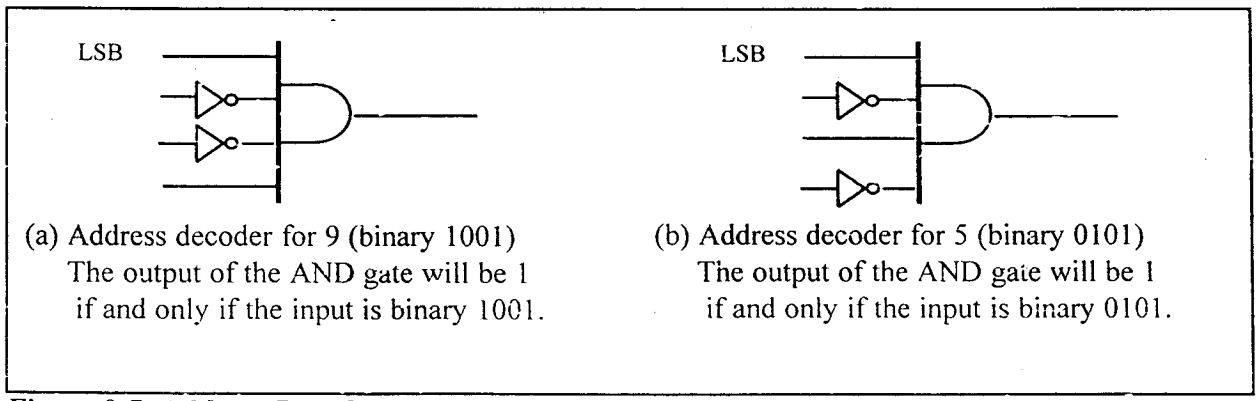


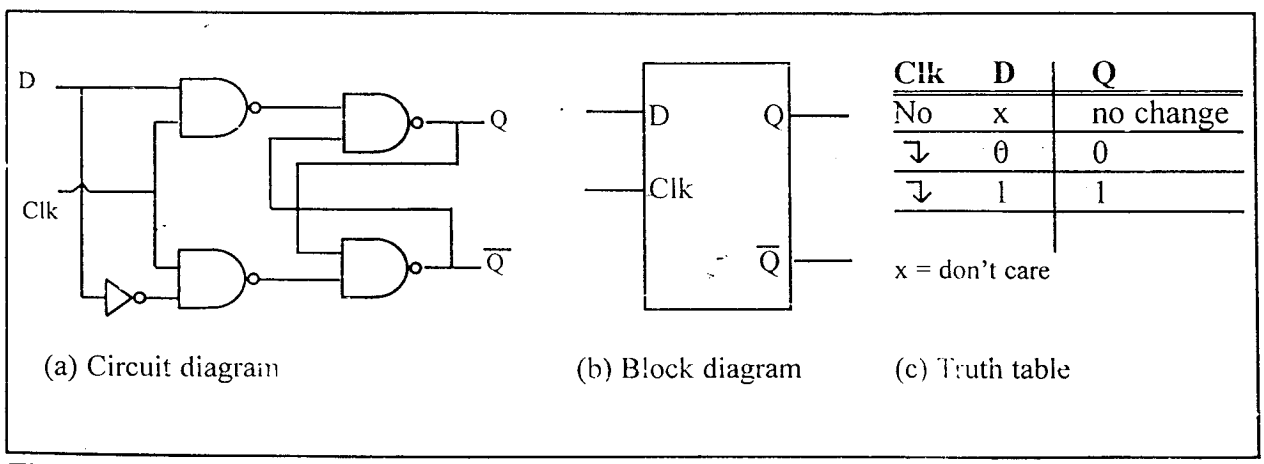
Figure 0-6. 3-Bit Adder Using 3 Full-Adders



(a) Address decoder for 9 (binary 1001)
The output of the AND gate will be 1
if and only if the input is binary 1001.

(b) Address decoder for 5 (binary 0101)
The output of the AND gate will be 1
if and only if the input is binary 0101.

Figure 0-7. Address Decoders



Clk	D	Q
No	x	no change
↓	0	0
↓	1	1

x = don't care

(a) Circuit diagram (b) Block diagram (c) Truth table

Figure 0-8. D Flip-Flops

Review Questions

1. The logical operation _____ gives a 1 output when all inputs are 1.
2. The logical operation _____ gives a 1 output when 1 or more of its inputs is 1.
3. The logical operation _____ is often used to compare if two inputs have the same value.
4. A _____ gate does not change the logic level of the input.
5. Name a common use for flip-flops.
6. An address _____ is used to identify a pre-determined binary address.

SECTION 0.3: INSIDE THE COMPUTER

In this section we provide an introduction to the organization and internal working of computers. The model used is generic, but the concepts discussed are applicable to all computers, including the IBM PC, PS/2, and compatibles. Before embarking on this subject, it will be helpful to review definitions of some of the most widely used terminology in computer literature, such as *K*, *mega*, *giga*, *byte*, *ROM*, *RAM*, and so on.

Some important terminology

One of the most important features of a computer is how much memory it has. Next we review terms used to describe amounts of memory in IBM PCs and compatibles. Recall from the discussion above that a *bit* is a binary digit that can have the value 0 or 1. A *byte* is defined as 8 bits. A *nibble* is half a byte, or 4 bits. A *word* is two bytes, or 16 bits. The display is intended to show the relative size of these units. Of course, they could all be composed of any combination of zeros and ones.

```
Bit                                   0
Nibble                             0000
Byte                          0000 0000
Word 0000 0000 0000 0000
```

A *kilobyte* is 2^{10} bytes, which is 1024 bytes. The abbreviation K is often used. For example, some floppy disks hold 356K bytes of data. A *megabyte*, or meg as some call it, is 2^{20} bytes. That is a little over 1 million bytes; it is exactly 1,048,576 bytes. Moving rapidly up the scale in size, a *gigabyte* is 2^{30} bytes (over 1 billion), and a *terabyte* is 2^{40} bytes (over 1 trillion). As an example of how some of these terms are used, suppose that a given computer has 16 megabytes of memory. That would be 16×2^{20}, or $2^4 \times 2^{20}$, which is 2^{24}. Therefore 16 megabytes is 2^{24} bytes.

Two types of memory commonly used in microcomputers are *RAM*, which stands for "random access memory" (sometimes called *read/write memory*), and *ROM*, which stands for "read-only memory." RAM is used by the computer for temporary storage of programs that it is running. That data is lost when the computer is turned off. For this reason, RAM is sometimes called *volatile memory*. ROM contains programs and information essential to operation of the computer. The information in ROM is permanent, cannot be changed by the user, and is not lost when the power is turned off. Therefore, it is called *nonvolatile memory*.

Internal organization of computers

The internal working of every computer can be broken down into three parts: CPU (central processing unit), memory , and I/O (input/output) devices (see Figure 0-9). The function of the CPU is to execute (process) information stored in memory. The function of I/O devices such as the keyboard and video monitor is to provide a means of communicating with the CPU. The CPU is connected to memory and I/O through strips of wire called a *bus*. The bus inside a computer carries information from place to place just as a street bus carries people from place to place. In every computer there are three types of buses: address bus, data bus, and control bus.

For a device (memory or I/O) to be recognized by the CPU, it must be assigned an address. The address assigned to a given device must be unique; no two devices are allowed to have the same address. The CPU puts the address (of course, in binary) on the address bus, and the decoding circuitry finds the device. Then the CPU uses the data bus either to get data from that device or to send data to it. The control buses are used to provide read or write signals to the device to indicate if the CPU is asking for information or sending it information. Of the three buses, the address bus and data bus determine the capability of a given CPU.

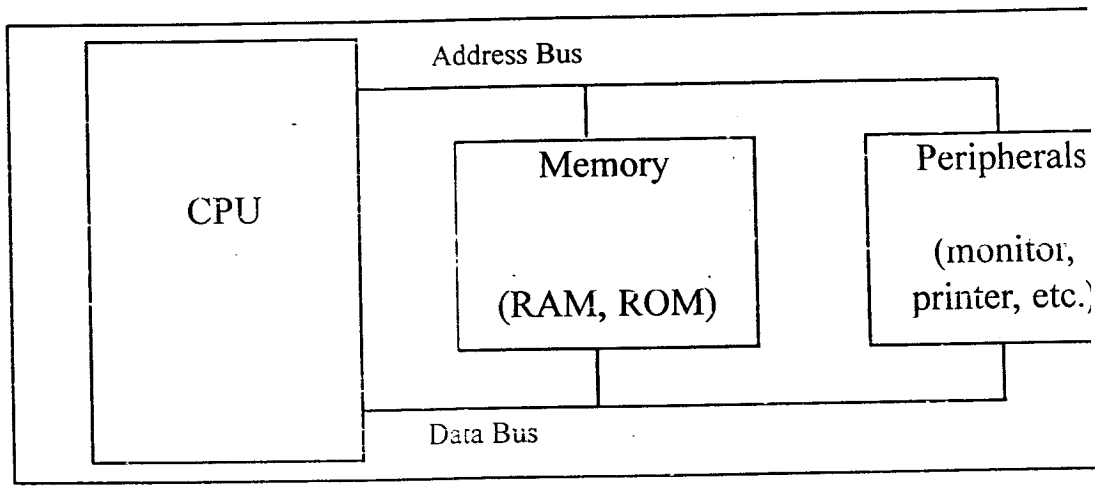


Figure 0-9: Inside the Computer

More about the data bus

Since data buses are used to carry information in and out of a CPU, the more data buses available, the better the CPU. If one thinks of data buses as highway lanes, it is clear that more lanes provide a better pathway between the CPU and its external devices (such as printers, RAM, ROM, etc.; see Figure 0-10). By the same token, that increase in the number of lanes increases the cost of construction. More data buses mean a more expensive CPU and computer. The average size of data buses in CPUs varies between 8 and 64. Early computers such as Apple 2 used an 8-bit data bus, while supercomputers such as Cray use a 64-bit data bus. Data buses are bidirectional, since the CPU must use them either to receive or to send data. The processing power of a computer is related to the size of its buses, since an 8-bit bus can send out 1 byte a time, but a 16-bit bus can send out 2 bytes at a time, which is twice as fast.

More about the address bus

Since the address bus is used to identify the devices and memory connected to the CPU, the more address buses available, the larger the number of devices that can be addressed. In other words, the number of address buses for a CPU determines the number of locations with which it can communicate. The number of locations is always equal to 2^x, where x is the number of address lines, regardless of the size of the data bus. For example, a CPU with 16 address lines can provide a total of 65,536 (2^{16}) or 64K bytes of addressable memory. Each location can have a maximum of 1 byte of data. This is due to the fact that all general-purpose microprocessor CPUs are what is called *byte addressable*. As another example, the IBM PC AT uses a CPU with 24 address lines and 16 data lines. In this case the total accessible memory is 16 megabytes (2^{24} = 16 megabytes). In this example there would be 2^{24} locations, and since each location is one byte, there would be 16 megabytes of memory. The address bus is a *unidirectional* bus, which means that the CPU uses the address bus only to send out addresses. To summarize: The total number of memory locations addressable by a given CPU is always equal to 2^x where x is the number of address bits, regardless of the size of the data bus.

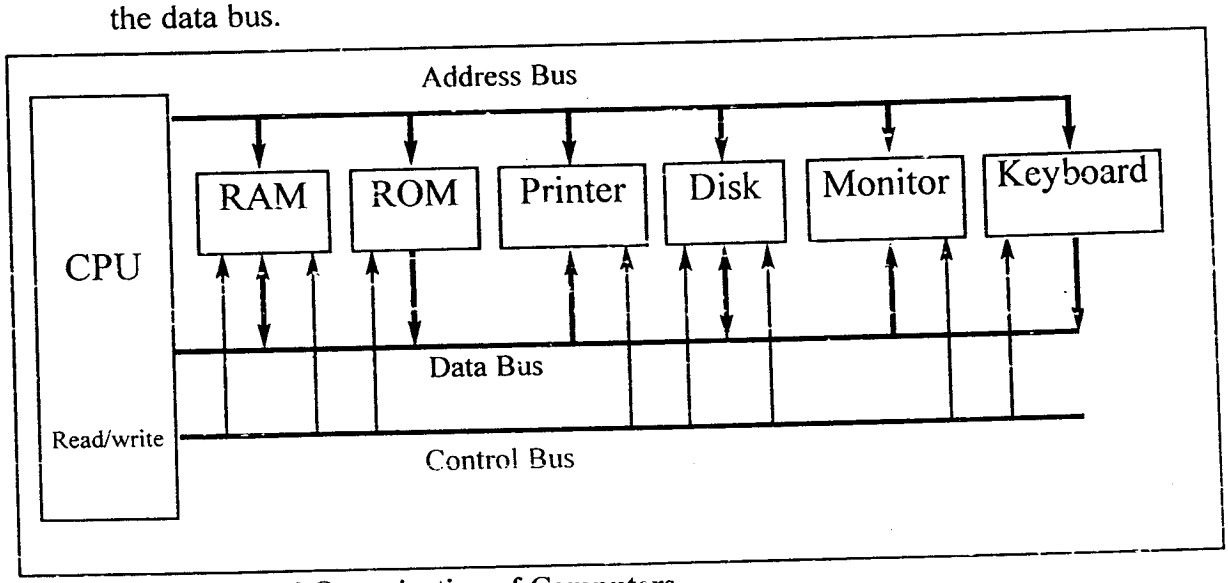


Figure 0-10: Internal Organization of Computers

CPU and its relation to RAM and ROM

For the CPU to process information, the data must be stored in RAM or ROM. The function of ROM in computers is to provide information that is fixed and permanent. This is information such as tables for character patterns to be displayed on the video monitor, or programs that are essential to the working of the computer, such as programs for testing and finding the total amount of RAM installed on the system, or programs to display information on the video monitor. In contrast, RAM is used to store information that is not permanent and can change with time, such as various versions of the operating system and application packages such as word processing or tax calculation packages. These programs are loaded into RAM to be processed by the CPU. The CPU cannot get the informa-

tion from the disk directly since the disk is too slow. In other words, the CPU gets the information to be processed, first from RAM (or ROM). Only if it is not there does the CPU seek it from a mass storage device such as a disk, and then it transfers the information to RAM. For this reason, RAM and ROM are sometimes referred to as *primary memory* and disks are called *secondary memory*. Figure 0-11 shows a block diagram of the internal organization of the PC.

Inside CPUs

A program stored in memory provides instructions to the CPU to perform an action. The action can simply be adding data such as payroll data or controlling a machine such as a robot. It is the function of the CPU to fetch these instructions from memory and execute them. To perform the actions of fetch and execute, all CPUs are equipped with resources such as the following:

1. Foremost among the resources at the disposal of the CPU are a number of *registers*. The CPU uses registers to store information temporarily. The information could be two values to be processed, or the address of the value needed to be fetched from memory. Registers inside the CPU can be 8-bit, 16-bit, 32-bit, or even 64-bit registers, depending on the CPU. In general, the more and bigger the registers, the better the CPU. The disadvantage of more and bigger registers is the increased cost of such a CPU.
2. The CPU also has what is called the *ALU* (arithmetic/logic unit). The ALU section of the CPU is responsible for performing arithmetic functions such as add, subtract, multiply, and divide, and logic functions such as AND, OR, and NOT.
3. Every CPU has what is called a *program counter*. The function of the program counter is to point to the address of the next instruction to be executed. As each instruction is executed, the program counter is incremented to point to the address of the next instruction to be executed. It is the contents of the program counter that are placed on the address bus to find and fetch the desired instruction. In the IBM PC, the program counter is a register called IP, or the instruction pointer.
4. The function of the *instruction decoder* is to interpret the instruction fetched into the CPU. One can think of the instruction decoder as a kind of dictionary, storing the meaning of each instruction and what steps the CPU should take upon receiving a given instruction. Just as a dictionary requires more pages the more words it defines, a CPU capable of understanding more instructions requires more transistors to design.

Internal working of computers

To demonstrate some of the concepts discussed above, a step-by-step analysis of the process a CPU would go through to add three numbers is given next. Assume that an imaginary CPU has registers called A, B, C, and D. It has an 8-bit data bus and a 16-bit address bus. Therefore, the CPU can access memory from addresses 0000 to FFFFH (for a total of 10000H locations). The action to be performed by the CPU is to put hexadecimal value 21 into register A, and then add to register A values 42H and 12H. Assume that the code for the CPU to move a

value to register A is 1011 0000 (B0H) and the code for adding a value to register A is 0000 0100 (04H). The necessary steps and code to perform them are as follows.

Action	Code	Data
Move value 21H into register A	B0H	21H
Add value 42H to register A	04H	42H
Add value 12H to register A	04H	12H

If the program to perform the actions listed above is stored in memory locations starting at 1400H, the following would represent the contents for each memory address location:

Memory address	Contents of memory address
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

The actions performed by the CPU to run the program above would be as follows:

1. The CPU's program counter can have a value between 0000 and FFFFH. The program counter must be set to the value 1400H, indicating the address of the first instruction code to be executed. After the program counter has been loaded with the address of the first instruction, the CPU is ready to execute.
2. The CPU puts 1400H on the address bus and sends it out. The memory circuitry finds the location while the CPU activates the READ signal, indicating to memory that it wants the byte at location 1400H. This causes the contents of memory location 1400H, which is B0, to be put on the data bus and brought into the CPU.
3. The CPU decodes the instruction B0 with the help of its instruction decoder dictionary. When it finds the definition for that instruction it knows it must bring into register A of the CPU the byte in the next memory location. Therefore, it commands its controller circuitry to do exactly that. When it brings in value 21H from memory location 1401, it makes sure that the doors of all registers are closed except register A. Therefore, when value 21H comes into the CPU it will go directly into register A. After completing one instruction, the program counter points to the address of the next instruction to be executed, which in this case is 1402H. Address 1402 is sent out on the address bus to fetch the next instruction.
4. From memory location 1402H it fetches code 04H. After decoding, the CPU knows that it must add to the contents of register A the byte sitting at the next address (1403). After it brings the value (in this case 42H) into the CPU, it provides the contents of register A along with this value to the ALU to perform the addition. It then takes the result of the addition from the ALU's output and

puts it in register A. Meanwhile the program counter becomes 1404, the address of the next instruction.

5. Address 1404H is put on the address bus and the code is fetched into the CPU, decoded, and executed. This code is again adding a value to register A. The program counter is updated to 1406H.
6. Finally, the contents of address 1406 are fetched in and executed. This HALT instruction tells the CPU to stop incrementing the program counter and asking for the next instruction. In the absence of the HALT, the CPU would continue updating the program counter and fetching instructions.

Now suppose that address 1403H contained value 04 instead of 42H. How would the CPU distinguish between data 04 to be added and code 04? Remember that code 04 for this CPU means move the next value into register A. Therefore, the CPU will not try to decode the next value. It simply moves the contents of the following memory location into register A, regardless of its value.

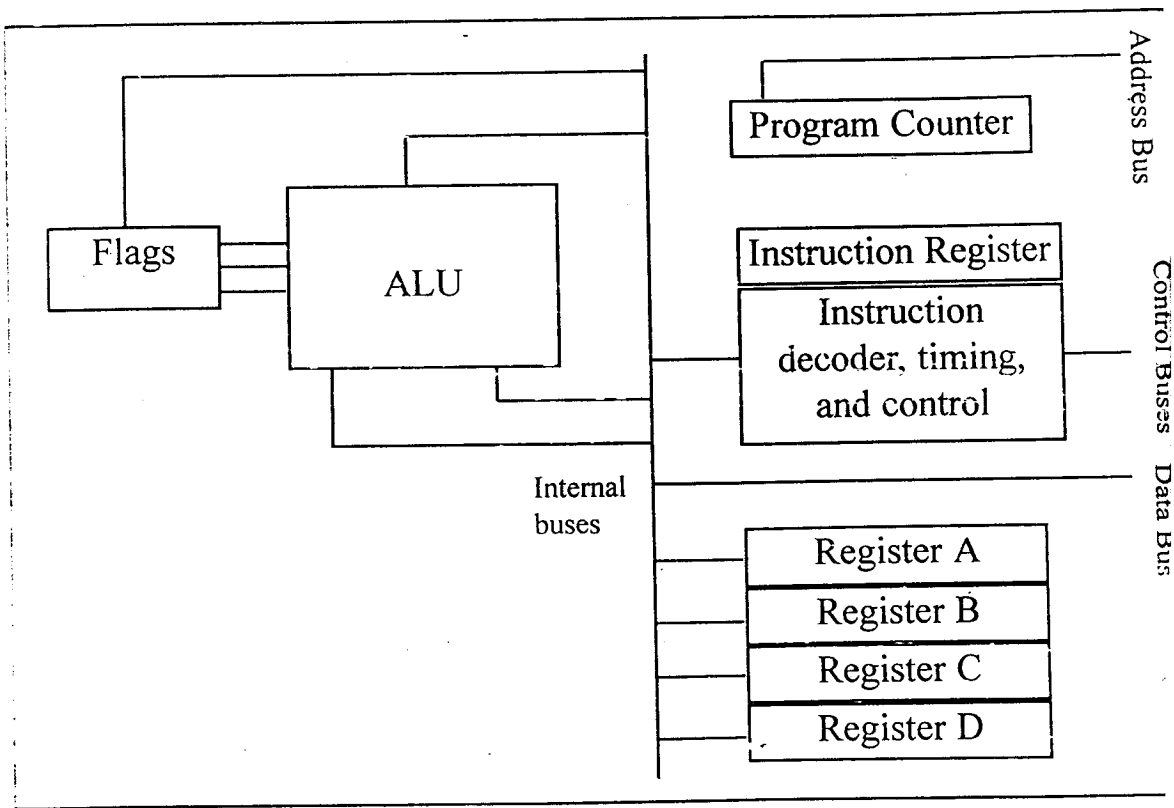


Figure 0-11: Internal Block Diagram of a CPU

Review Questions

1. How many bytes is 24 kilobytes?
2. What does "RAM" stand for? How is it used in computer systems?
3. What does "ROM" stand for? How is it used in computer systems?
4. Why is RAM called volatile memory?
5. List the three major components of a computer system.
6. What does "CPU" stand for? Explain its function in a computer.

7. List the three types of buses found in computer systems and state briefly the purpose of each type of bus.
8. State which of the following is unidirectional and which is bidirectional.
   (a) data bus (b) address bus
9. If an address bus for a given computer has 16 lines, what is the maximum amount of memory it can access?
10. What does "ALU" stand for? What is its purpose?
11. How are registers used in computer systems?
12. What is the purpose of the program counter?
13. What is the purpose of the instruction decoder?

SUMMARY

The binary number system represents all numbers with a combination of the two binary digits, 0 and 1. The use of binary systems is necessary in digital computers because only two states can be represented: on or off. Any binary number can be coded directly into its hexadecimal equivalent for the convenience of humans. Converting from binary/hex to decimal, and vice versa, is a straightforward process that becomes easy with practice. The ASCII code is a binary code used to represent alphanumeric data internally in the computer. It is frequently used in peripheral devices for input and/or output.

The logic gates AND, OR, and Inverter are the basic building blocks of simple circuits. NAND, NOR, and XOR gates are also used to implement circuit design. Diagrams of half-adders and full-adders were given as examples of the use of logic gates for circuit design. Decoders are used to detect certain addresses. Flip-flops are used to latch in data until other circuits are ready for it.

The major components of any computer system are the CPU, memory, and I/O devices. "Memory" refers to temporary or permanent storage of data. In most systems, memory can be accessed as bytes or words. The terms *kilobyte*, *megabyte*, *gigabyte*, and *terabyte* are used to refer to large numbers of bytes. There are two main types of memory in computer systems: RAM and ROM. RAM (random access memory) is used for temporary storage of programs and data. ROM (read-only memory) is used for permanent storage of programs and data that the computer system must have in order to function. All components of the computer system are under the control of the CPU. Peripheral devices such as I/O (input/output) devices allow the CPU to communicate with humans or other computer systems. There are three types of buses in computers: address, control, and data. Control buses are used by the CPU to direct other devices. The address bus is used by the CPU to locate a device or a memory location. Data buses are used to send information back and forth between the CPU and other devices.

Finally, this chapter gave an overview of digital logic.

PROBLEMS

SECTION 0.1: NUMBERING AND CODING SYSTEMS

1. Convert the following decimal numbers to binary.
   (a) 12 (b) 123 (c) 63 (d) 128 (e) 1000
2. Convert the following binary numbers to decimal.
   (a) 100100 (b) 1000001 (c) 11101 (d) 1010 (e) 00100010
3. Convert the values in Problem 2 to hexadecimal.
4. Convert the following hex numbers to binary and decimal.
   (a) 2B9H (b) F44H (c) 912H (d) 2BH (e) FFFFH
5. Convert the values in Problem 1 to hex.
6. Find the 2's complement of the following binary numbers.
   (a) 1001010 (b) 111001 (c) 10000010 (d) 111110001
7. Add the following hex values.
   (a) 2CH+3FH (b) F34H+5D6H (c) 20000H+12FFH (d) FFFFH+2222H
8. Perform hex subtraction for the following.
   (a) 24FH-129H (b) FE9H-5CCH (c) 2FFFFH-FFFFFH (d) 9FF25H-4DD99H
9. Show the ASCII codes for numbers 0, 1, 2, 3, ..., 9 in both hex and binary.
10. Show the ASCII code (in hex) for the following string:
    "U.S.A. is a country" CR,LF
    "in North America" CR,LF
    CR is carriage return
    LF is line feed

SECTION 0.2: DIGITAL PRIMER

11. Draw a 3-input OR gate using a 2-input OR gate.
12. Show the truth table for a 3-input OR gate.
13. Draw a 3-input AND gate using a 2-input AND gate.
14. Show the truth table for a 3-input AND gate.
15. Design a 3-input XOR gate with a 2-input XOR gate. Show the truth table for a 3-input XOR.
16. List the truth table for a 3-input NAND.
17. List the truth table for a 3-input NOR.
18. Show the decoder for binary 1100.
19. Show the decoder for binary 11011.
20. List the truth table for a D-FF.

SECTION 0.3: INSIDE THE COMPUTER

21. Answer the following:
    (a) How many nibbles are 16 bits?
    (b) How many bytes are 32 bits?
    (c) If a word is defined as 16 bits, how many words is a 64-bit data item?
    (d) What is the exact value (in decimal) of 1 meg?

(e) How many K is 1 meg?
(f) What is the exact value (in decimal) of 1 giga?
(g) How many K is 1 giga?
(h) How many meg is 1 giga?
(i) If a given computer has a total of 8 megabytes of memory, how many bytes (in decimal) is this? How many kilobytes is this?

22. A given mass storage device such as a hard disk can store 2 gigabytes of information. Assuming that each page of text has 25 rows and each row has 80 columns of ASCII characters (each character = 1 byte), approximately how many pages of information can this disk store?
23. In a given byte-addressable computer, memory locations 10000H to 9FFFFH are available for user programs. The first location is 10000H and the last location is 9FFFFH. Calculate the following:
(a) The total number of bytes available (in decimal)
(b) The total number of kilobytes (in decimal)
24. A given computer has a 32-bit data bus. What is the largest number that can be carried into the CPU at a time?
25. Below are listed several computers with their data bus widths. For each computer, list the maximum value that can be brought into the CPU at a time (in both hex and decimal).
(a) Apple 2 with an 8-bit data bus
(b) IBM PS/2 with a 16-bit data bus
(c) IBM PS/2 model 80 with a 32-bit data bus
(d) CRAY supercomputer with a 64-bit data bus
26. Find the total amount of memory, in the units requested, for each of the following CPUs, given the size of the address buses.
(a) 16-bit address bus (in K)
(b) 24-bit address bus (in meg)
(c) 32-bit address bus (in megabytes and gigabytes)
(d) 48-bit address bus (in megabytes, gigabytes, and terabytes)
27. Regarding the data bus and address bus, which is unidirectional and which is bidirectional?
28. Which register of the CPU holds the address of the instruction to be fetched?
29. Which section of the CPU is responsible for performing addition?
30. List the three bus types present in every CPU.

ANSWERS TO REVIEW QUESTIONS

SECTION 0.1: NUMBERING AND CODING SYSTEMS

1. Computers use the binary system because each bit can have one of two voltage levels: on and off.
2. $34_{10} = 100010_2 = 22_{16}$
3. $110101_2 = 35_{16} = 53_{10}$
4. 1110001
5. 010100
6. 461
7. 275
8. 38 30 78 38 36 20 43 50 55 73

SECTION 0.2: DIGITAL PRIMER

1. AND
2. OR
3. XOR
4. Buffer
5. Storing data
6. Decoder

SECTION 0.3: INSIDE THE COMPUTER

1. 24,576
2. Random access memory; it is used for temporary storage of programs that the CPU is running, such as the operating system, word processing programs, etc.
3. Read-only memory; it is used for permanent programs such as those that control the keyboard, etc.
4. The contents of RAM are lost when the computer is powered off.
5. The CPU, memory, and I/O devices
6. Central processing unit; it can be considered the "brain" of the computer; it executes the programs and controls all other devices in the computer.
7. The address bus carries the location (address) needed by the CPU; the data bus carries information in and out of the CPU; the control bus is used by the CPU to send signals controlling I/O devices.
8. (a) bidirectional (b) unidirectional
9. 64K, or 65,536 bytes
10. Arithmetic/logic unit; it performs all arithmetic and logic operations.
11. It is for temporary storage of information.
12. It holds the address of the next instruction to be executed.
13. It tells the CPU what steps to perform for each instruction.

CHAPTER 1

THE 8051 MICROCONTROLLERS

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **Compare and contrast microprocessors and microcontrollers**
- **Describe the advantages of microcontrollers for some applications**
- **Explain the concept of embedded systems**
- **Discuss criteria to consider in choosing a microcontroller**
- **Explain the variations of speed, packaging, memory, and cost per unit and how these affect choosing a microcontroller**
- **Compare and contrast the various members of the 8051 family**
- **Compare 8051 microcontrollers offered by various manufacturers**

This chapter begins with a discussion of the role and importance of microcontrollers in everyday life. In Section 1.1 we also discuss criteria to consider in choosing a microcontroller, as well as the use of microcontrollers in the embedded market. Section 1.2 covers various members of the 8051 family such as the 8052 and 8031, and their features. In addition, we discuss various versions of the 8051 such as the 8751, AT89C51, and DS5000.

SECTION 1.1: MICROCONTROLLERS AND EMBEDDED PROCESSORS

In this section we discuss the need for microcontrollers and contrast them with general-purpose microprocessors such as the Pentium and other x86 microprocessors. We also look at the role of microcontrollers in the embedded market. In addition, we provide some criteria on how to choose a microcontroller.

Microcontroller versus general-purpose microprocessor

What is the difference between a microprocessor and microcontroller? By microprocessor is meant the general-purpose microprocessors such as Intel's x86 family (8086, 80286, 80386, 80486, and the Pentium) or Motorola's 680x0 family (68000, 68010, 68020, 68030, 68040, etc.). These microprocessors contain no RAM, no ROM, and no I/O ports on the chip itself. For this reason, they are commonly referred to as *general-purpose microprocessors*.

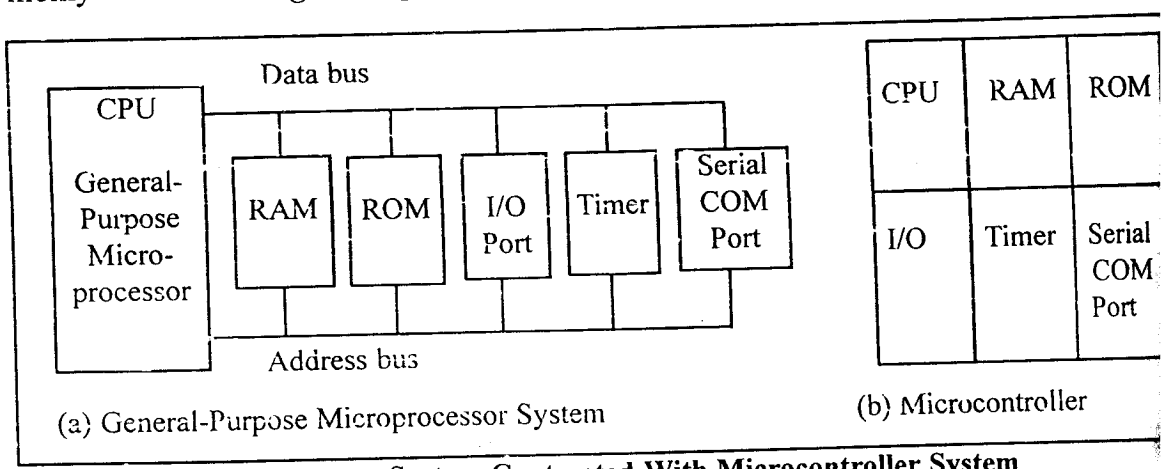


Figure 1-1. Microprocessor System Contrasted With Microcontroller System

A system designer using a general-purpose microprocessor such as the Pentium or the 68040 must add RAM, ROM, I/O ports, and timers externally to make them functional. Although the addition of external RAM, ROM, and I/O ports makes these systems bulkier and much more expensive, they have the advantage of versatility such that the designer can decide on the amount of RAM, ROM, and I/O ports needed to fit the task at hand. This is not the case with microcontrollers. A microcontroller has a CPU (a microprocessor) in addition to a fixed amount of RAM, ROM, I/O ports, and a timer all on a single chip. In other words, the processor, the RAM, ROM, I/O ports, and timer are all embedded together on one chip; therefore, the designer cannot add any external memory, I/O, or timer to it. The fixed amount of on-chip ROM, RAM, and number of I/O ports in microcontrollers makes them ideal for many applications in which cost and space are

Home
Appliances
Intercom
Telephones
Security systems
Garage door openers
Answering machines
Fax machines
Home computers
TVs
Cable TV tuner
VCR
Camcorder
Remote controls
Video games
Cellular phones
Musical instruments
Sewing machines
Lighting control
Paging
Camera
Pinball machines
Toys
Exercise equipment
Office
Telephones
Computers
Security systems
Fax machine
Microwave
Copier
Laser printer
Color printer
Paging
Auto
Trip computer
Engine control
Air bag
ABS
Instrumentation
Security system
Transmission control
Entertainment
Climate control
Cellular phone
Keyless entry

Table 1-1: Some Embedded Products Using Microcontrollers

critical. In many applications, for example a TV remote control, there is no need for the computing power of a 486 or even an 8086 microprocessor. In many applications, the space it takes, the power it consumes, and the price per unit are much more critical considerations than the computing power. These applications most often require some I/O operations to read signals and turn on and off certain bits. For this reason some call these processors IBP, "itty-bitty processors" (see "Good Things in Small Packages Are Generating Big Product Opportunities" by Rick Grehan, BYTE magazine, September 1994; www.byte.com, for an excellent discussion of microcontrollers).

It is interesting to note that some microcontroller manufacturers have gone as far as integrating an ADC (analog-to-digital converter) and other peripherals into the microcontroller.

Microcontrollers for embedded systems

In the literature discussing microprocessors, we often see the term *embedded system*. Microprocessors and microcontrollers are widely used in embedded system products. An embedded product uses a microprocessor (or microcontroller) to do one task and one task only. A printer is an example of embedded system since the processor inside it performs one task only; namely, getting the data and printing it. Contrast this with a Pentium-based PC (or any x86 IBM-compatible PC). A PC can be used for any number of applications such as word processor, print-server, bank teller terminal, video game player, network server, or internet terminal. Software for a variety of applications can be loaded and run. Of course the reason a PC can perform myriad tasks is that it has RAM memory and an operating system that loads the application software into RAM and lets the CPU run it. In an embedded system, there is only one application software that is typically burned into ROM. An x86 PC contains or is connected to various embedded products such as the keyboard, printer, modem, disk controller, sound card, CD-ROM driver, mouse, and so on. Each one of these peripherals has a microcontroller inside it that performs only one task. For example, inside every mouse there is a microcontroller to perform the task of finding the mouse position and sending it to the PC. Table 1-1 lists some embedded products.

X86 PC embedded applications

Although microcontrollers are the preferred choice for many embedded systems, there are times that a microcontroller is inadequate for the task. For this reason, in recent years many manufacturers of general-purpose microprocessors such as Intel, Motorola, AMD (Advanced Micro Devices, Inc.), and Cyrix (now a division of National Semiconductor, Inc.) have targeted

their microprocessor for the high end of the embedded market. While Intel, AMD, and Cyrix push their x86 processors for both the embedded and desk-top PC markets, Motorola is determined to keep the 68000 family alive by targeting it mainly for the high end of embedded systems now that Apple no longer uses the 680x0 in their Macintosh. In the early 1990s Apple computer began using Power PC microprocessors (604, 603, 620, etc.) in place of the 680x0 for the Macintosh. The Power PC microprocessor is a joint venture between IBM and Motorola, and is targeted for the high end of the embedded market as well as the PC market. It must be noted that when a company targets a general-purpose microprocessor for the embedded market it optimizes the processor used for embedded systems. For this reason these processors are often called *high-end embedded processors*. Very often the terms *embedded processor* and *microcontroller* are used interchangeably.

One of the most critical needs of an embedded system is to decrease power consumption and space. This can be achieved by integrating more functions into the CPU chip. All the embedded processors based on the x86 and 680x0 have low power consumption in addition to some forms of I/O, COM port, and ROM all on a single chip. In high-performance embedded processors, the trend is to integrate more and more functions on the CPU chip and let the designer decide which features he/she wants to use. This trend is invading PC system design as well. Normally, in designing the PC motherboard we need a CPU plus a chip-set containing I/O, a cache controller, a flash ROM containing BIOS, and finally a secondary cache memory. New designs are emerging in industry. For example, Cyrix has announced that it is working on a chip that contains the entire PC, except for DRAM. In other words, we are about to see an entire computer on a chip.

Currently, because of MS-DOS and Windows standardization many embedded systems are using x86 PCs. In many cases using x86 PCs for the high-end embedded applications not only saves money but also shortens development time since there is a vast library of software already written for the DOS and Windows platforms. The fact that Windows is a widely used and well understood platform means that developing a Windows-based embedded product reduces the cost and shortens the development time considerably.

Choosing a microcontroller

There are four major 8-bit microcontrollers. They are: Motorola's 6811, Intel's 8051, Zilog's Z8, and PIC 16X from Microchip Technology. Each of the above microcontrollers has a unique instruction set and register set; therefore, they are not compatible with each other. Programs written for one will not run on the others. There are also 16-bit and 32-bit microcontrollers made by various chip makers. With all these different microcontrollers, what criteria do designers consider in choosing one? Three criteria in choosing microcontrollers are as follows: (1) meeting the computing needs of the task at hand efficiently and cost effectively, (2) availability of software development tools such as compilers, assemblers, and debuggers, and (3) wide availability and reliable sources of the microcontroller. Next we elaborate further on each of the above criteria.

Criteria for choosing a microcontroller

1. The first and foremost criterion in choosing a microcontroller is that it must meet the task at hand efficiently and cost effectively. In analyzing the needs of a microcontroller-based project, we must first see whether an 8-bit, 16-bit, or 32-bit microcontroller can best handle the computing needs of the task most effectively. Among other considerations in this category are:
   (a) Speed. What is the highest speed that the microcontroller supports?
   (b) Packaging. Does it come in 40-pin DIP (dual inline package) or a QFP (quad flat package), or some other packaging format? This is important in terms of space, assembling, and prototyping the end product.
   (c) Power consumption. This is especially critical for battery-powered products.
   (d) The amount of RAM and ROM on chip.
   (e) The number of I/O pins and the timer on the chip.
   (f) How easy it is to upgrade to higher-performance or lower power-consumption versions.
   (g) Cost per unit. This is important in terms of the final cost of the product in which a microcontroller is used. For example, there are microcontrollers that cost 50 cents per unit when purchased 100,000 units at a time.

2. The second criterion in choosing a microcontroller is how easy it is to develop products around it. Key considerations include the availability of an assembler, debugger, a code-efficient C language compiler, emulator, technical support, and both in-house and outside expertise. In many cases, third-party vendor (that is, a supplier other than the chip manufacturer) support for the chip is as good as, if not better than, support from the chip manufacturer.

3. The third criterion in choosing a microcontroller is its ready availability in needed quantities both now and in the future. For some designers this is even more important than the first two criteria. Currently, of the leading 8-bit microcontrollers, the 8051 family has the largest number of diversified (multiple source) suppliers. By supplier is meant a producer besides the originator of the microcontroller. In the case of the 8051, which was originated by Intel, several companies also currently produce (or have produced in the past) the 8051. These companies include: Intel, Atmel, Philips/Signetics, AMD, Siemens, Matra, and Dallas Semiconductor.

Table 1-2: Some Companies Producing a Member of the 8051 Family

Company	Web Site
Intel	www.intel.com/design/mcs51
Atmel	www.atmel.com
Philips/Signetics	www.semiconductors.philips.com
Siemens	www.sci.siemens.com
Dallas Semiconductor	www.dalsemi.com

It should be noted that Motorola, Zilog, and Microchip Technology have all dedicated massive resources to ensure wide and timely availability of their product since their product is stable, mature, and single sourced. In recent years they also have begun to sell the ASIC library cell of the microcontroller.

Review Questions

1. True or false. Microcontrollers are normally less expensive than microprocessors.
2. When comparing a system board based on a microcontroller and a general-purpose microprocessor, which one is cheaper?
3. A microcontroller normally has which of the following devices on-chip?
   (a) RAM (b) ROM (c) I/O (d) all of the above
4. A general-purpose microprocessor normally needs which of the following devices to be attached to it?
   (a) RAM (b) ROM (c) I/O (d) all of the above
5. An embedded system is also called a dedicated system. Why?
6. What does the term *embedded system* mean?
7. Why does having multiple sources of a given product matter?

SECTION 1.2: OVERVIEW OF THE 8051 FAMILY

In this section we first look at the various members of the 8051 family of microcontrollers and their internal features. Plus we see who are the different manufacturers of the 8051 and what kind of products they offer.

A brief history of the 8051

In 1981, Intel Corporation introduced an 8-bit microcontroller called the 8051. This microcontroller had 128 bytes of RAM, 4K bytes of on-chip ROM, two timers, one serial port, and four ports (each 8-bits wide) all on a single chip. At the time it was also referred to as a "system on a chip." The 8051 is an 8-bit processor, meaning that the CPU can work on only 8 bits of data at a time. Data larger than 8 bits has to be broken into 8-bit pieces to be processed by the CPU. The 8051 has a total of four I/O ports, each 8 bits wide. See Figure 1-2. Although the 8051 can have a maximum of 64K bytes of on-chip ROM, many manufacturers have put only 4K bytes on the chip. This will be discussed in more detail later.

The 8051 became widely popular after Intel allowed other manufacturers to make and market any flavor of the 8051 they please with the condition that they remain code-compatible with the 8051. This has led to many versions of the 8051 with different speeds and amounts of on-chip ROM marketed by more than half a dozen manufacturers. Next we review some of them. It is important to note that although there are different flavors of the 8051 in terms of speed and amount of on-chip ROM, they are all compatible with the original 8051 as far as the instructions are concerned. This means that if you write your program for one, it will run on any one of them regardless of the manufacturer.

Table 1-3: Features of the 8051

Feature	Quantity
ROM	4K bytes
RAM	128 bytes
Timer	2
I/O pins	32
Serial port	1
Interrupt sources	6

Note: ROM amount indicates on-chip program space.

8051 microcontroller

The 8051 is the original member of the 8051 family. Intel refers to it as MCS-51. Table 1-3 shows the main features of the 8051.

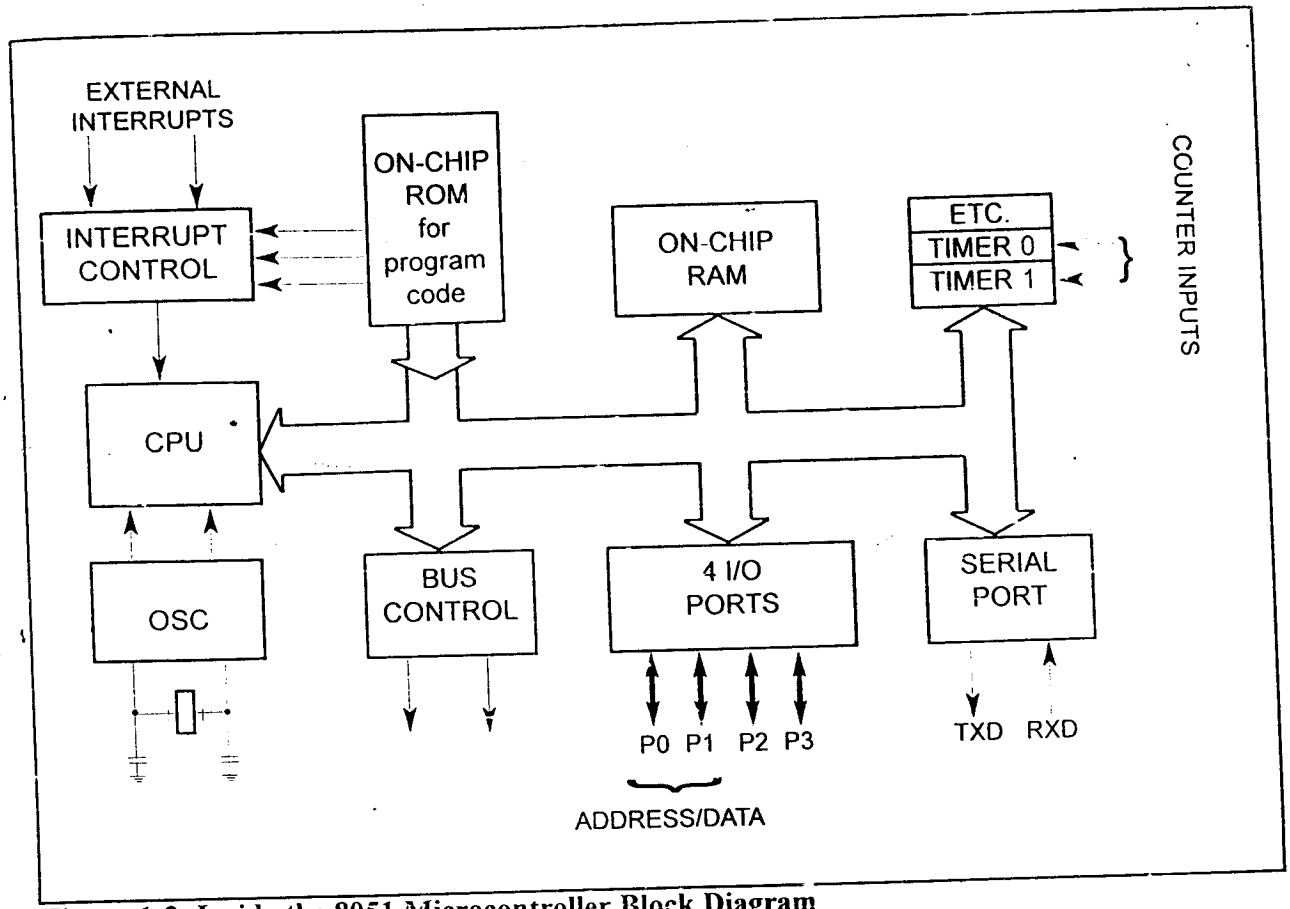


Figure 1-2. Inside the 8051 Microcontroller Block Diagram

Other members of the 8051 family

There are two other members in the 8051 family of microcontrollers. They are the 8052 and the 8031.

8052 microcontroller

The 8052 is another member of the 8051 family. The 8052 has all the standard features of the 8051 in addition to an extra 128 bytes of RAM and an extra timer. In other words, the 8052 has 256 bytes of RAM and 3 timers. It also has 8K bytes of on-chip program ROM instead of 4K bytes. See Table 1-4.

Table 1-4: Comparison of 8051 Family Members

Feature	8051	8052	8031
ROM (on-chip program space in bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

As can be seen from Table 1-4, the 8051 is a subset of the 8052; therefore, all programs written for the 8051 will run on the 8052, but the reverse is not true.

8031 microcontroller

Another member of the 8051 family is the 8031 chip. This chip is often referred to as a ROM-less 8051 since it has 0K bytes of on-chip ROM. To use this chip you must add external ROM to it. This external ROM must contain the program that the 8031 will fetch and execute. Contrast that to the 8051 in which the on-chip ROM contains the program to be fetched and executed but is limited to only 4K bytes of code. The ROM containing the program attached to the 8031 can be as large as 64K bytes. In the process of adding external ROM to the 8031, you lose two ports. That leaves only 2 ports (of the 4 ports) for I/O operations. To solve this problem, you can add external I/O to the 8031. Interfacing the 8031 with memory and I/O ports such as the 8255 chip is discussed in Chapter 14. There are also various speed versions of the 8031 available from different companies.

Various 8051 microcontrollers

Although the 8051 is the most popular member of the 8051 family, you will not see "8051" in the part number. This is because the 8051 is available in different memory types, such as UV-EPROM, flash, and NV-RAM, all of which have different part numbers. A discussion of the various types of ROM will be given in Chapter 14. The UV-EPROM version of the 8051 is the 8751. The flash ROM version is marketed by many companies including Atmel Corp. The Atmel Flash 8051 is called AT89C51. The NV-RAM version of the 8051 made by Dalias Semiconductor is called DS5000. There is also the OTP (one-time programmable) version of the 8051 made by various manufacturers. Next we discuss briefly each of the above chips and describe applications where they are used.

8751 microcontroller

This 8751 chip has only 4K bytes of on-chip UV-EPROM. To use this chip for development requires access to a PROM burner, as well as a UV-EPROM eraser to erase the contents of UV-EPROM inside the 8751 chip before you can program it again. Due to the fact that the on-chip ROM for the 8751 is UV-EPROM, it takes around 20 minutes to erase the 8751 before it can be programmed again. This has led many manufacturers to introduce flash and NV-RAM versions of the 8051 as we will discuss next. There are also various speed versions of the 8751 available from different companies.

AT89C51 from Atmel Corporation

This popular 8051 chip has on-chip ROM in the form of flash memory. This is ideal for fast development since flash memory can be erased in seconds compared to the twenty minutes or more needed for the 8751. For this reason the AT89C51 is used in place of the 8751 to eliminate the waiting time needed to erase the chip and thereby speed up the development time. To use the AT89C51 to develop a microcontroller-based system requires a ROM burner that supports flash memory; however, a ROM eraser is not needed. Notice that in flash memory you must erase the entire contents of ROM in order to program it again. This erasing of flash is done by the PROM burner itself and this is why a separate eraser is not needed. To eliminate the need for a PROM burner Atmel is working on a version of the AT89C51 that can be programmed via the serial COM port of an IBM PC.

Table 1-5: Versions of 8051 From Atmel (All ROM Flash)

Part Number	ROM	RAM	I/O pins	Timer	Interrupt	V_{CC}	Packaging
AT89C51	4K	128	32	2	6	5V	40
AT89LV51	4K	128	32	2	6	3V	40
AT89C1051	1K	64	15	1	3	3V	20
AT89C2051	2K	128	15	2	6	3V	20
AT89C52	8K	128	32	3	8	5V	40
AT89LV52	8K	128	32	3	8	3V	40

Note: "C" in the part number indicates CMOS.

There are various speed and packaging versions of the above products. See Table 1-6. For example, notice AT89C51-12PC where "C" before the 51 is for CMOS, which has a low power consumption, "12" indicates 12 MHz, "P" is for plastic DIP package, and "C" is for commercial (vs. "M" for military). Often, the AT89C51-12PC is ideal for many student projects.

Table 1-6: Various Speeds of 8051 From Atmel

Part Number	Speed	Pins	Packaging	Use
AT89C51-12PC	12 MHz	40	DIP plastic	commercial
AT89C51-16PC	16 MHz	40	DIP plastic	commercial
AT89C51-20PC	20 MHz	40	DIP plastic	commercial

DS5000 from Dallas Semiconductor

Another popular version of the 8051 is the DS5000 chip from Dallas Semiconductor. The on-chip ROM for the DS5000 is in the form of NV-RAM. The read/write capability of NV-RAM allows the program to be loaded into the on-chip ROM while it is in the system. This can be done even via the serial port of an IBM PC. This in-system program loading of DS5000 via a PC serial port makes it an ideal home development system. Another advantage of NV-RAM is the ability to change the ROM contents one byte at a time. Contrast this with UV-EPROM and flash memory in which the entire ROM must be erased before it is programmed again.

Table 1-7: Versions of 8051 From Dallas Semiconductor's Soft Microcontroller

Part Number	ROM	RAM	I/O pins	Timers	Interrupts	V_{CC}	Packaging
DS5000-8	8K	128	32	2	6	5V	40
DS5000-32	32K	128	32	2	6	5V	40
DS5000T-8	8K	128	32	2	6	5V	40
DS5000T-8	32K	128	32	2	6	5V	40

Notes: All ROM are NV-RAM.
"T" means it has a real-time clock.

Notice that the real-time clock (RTC) is different from the timer. The real-time clock generates and keeps the time of day (hr-min-sec) and date (yr-mon-day) even when the power is off.

There are various speed and packaging versions of the DS5000 as shown in Table 1-8. For example, DS5000-8-8 has 8K NV-RAM and a speed of 8MHz. Often the DS5000-8-12 (or DS5000T-8-12) is ideal for many student projects.

Table 1-8: Versions of 8051 From Dallas Semiconductor

Part Number	NV-RAM	Speed	
DS5000-8-8	8K	8 MHz	
DS5000-8-12	8K	12 MHz	
DS5000-32-8	32K	8 MHz	
DS5000T-32-8	32K	8 MHz	(with RTC)
DS5000-32-12	32K	12 MHz	
DS5000T-8-12	8K	12 MHz	(with RTC)

OTP version of the 8051

There are also OTP (one-time-programmable) versions of the 8051 available from different sources. Flash and NV-RAM versions are typically used for product development. When a product is designed and absolutely finalized, the OTP version of the 8051 is used for mass production since it is much cheaper in terms of price per unit.

8051 family from Philips

Another major producer of the 8051 family is Philips Corporation. Indeed, they have one of the largest selections of 8051 microcontrollers. Many of their products include features such as A-to-D converters, D-to-A converters, extended I/O, and both OTP and flash.

Review Questions

1. Name three features of the 8051.
2. What is the major difference between the 8051 and 8052 microcontrollers?
3. Give the size of RAM in each of the following.
   (a) 8051 (b) 8052 (c) 8031
4. Give the size of the on-chip ROM in each of the following.
   (a) 8051 (b) 8052 (c) 8031
5. The 8051 is a(n) _____-bit microprocessor.
6. State a major difference between the 8751, the AT89C51 and the DS5000.
7. List additional features introduced in the DS5000T that are not present in the DS5000.
8. True or false. The AT89C51-12PC chip has a DIP package.
9. The AT89C51-12PC chip can handle a maximum frequency of ____ MHz.
10. The DS5000-32 has _____ K bytes of on-chip NV-RAM for programs.

SUMMARY

This chapter discussed the role and importance of microcontrollers in everyday life. Microprocessors and microcontrollers were contrasted and compared. We discussed the use of microcontrollers in the embedded market. We also discussed criteria to consider in choosing a microcontroller such as speed, memory, I/O, packaging, and cost per unit. The second section of this chapter described various family members of the 8051, such as the 8052 and 8031, and their features. In addition, we discussed various versions of the 8051 such as the AT89C51 and DS5000, which are marketed by suppliers other than Intel.

PROBLEMS

SECTION 1.1: MICROCONTROLLERS AND EMBEDDED PROCESSORS

1. True or False. A general-purpose microprocessor has on-chip ROM.
2. True or False. A microcontroller has on-chip ROM.
3. True or False. A microcontroller has on-chip I/O ports.
4. True or False. A microcontroller has a fixed amount of RAM on the chip.
5. What components are normally put together with the microcontroller into a single chip?
6. Intel's Pentium chips used in Windows PCs need external _____ and _____ chips to store data and code.
7. List three embedded products attached to a PC.
8. Why would someone want to use an x86 as an embedded processor?
9. Give the name and the manufacturer of some of the most widely used 8-bit microcontrollers
10. In Question 9, which one has the most manufacture sources?
11. In a battery-based embedded product, what is the most important factor in choosing a microcontroller?
12. In an embedded controller with on-chip ROM, why does the size of the ROM matter?
13. In choosing a microcontroller, how important is it to have a multiple source for that chip?
14. What does the term "third-party support" mean?
15. If a microcontroller architecture has both 8-bit and 16-bit versions, which of the following statements is true.
    (a) The 8-bit software will run on the 16-bit system.
    (b) The 16-bit software will run on the 8-bit system.

SECTION 1.2: OVERVIEW OF THE 8051 FAMILY

16. The 8751 has _____ bytes of on-chip ROM.
17. The AT89C51 has _____ bytes of on-chip RAM.
18. The 8051 has ___ on-chip timer(s).

19. The 8052 has ____bytes of on-chip RAM.
20. The ROMless version of the 8051 uses _____ as the part number.
21. The 8051 family has ____ pins for I/O.
22. The 8051 family has circuitry to support _____ serial ports.
23. The 8751 on-chip ROM is of type _____.
24. The AT8951 on-chip ROM is of type _____.
25. The DS5000 on-chip ROM is of type _____.
26. Give the speed and package type for the following chips.
    (a) AT89C51-16PC (b) DS5000-8-12
27. In Question 26, give the amount and type of on-chip ROM.
28. Of the 8051 family, which version is the most cost effective if you are u
    million of them in an embedded product?
29. What is the difference between the 8031 and 8051?
30. Of the 8051 microcontrollers, which one is the best for a home develo
    environment? (You do not have access to a ROM burner).

ANSWERS TO REVIEW QUESTIONS

SECTION 1.1: MICROCONTROLLERS AND EMBEDDED PROCESSORS

1. True 2. A microcontroller based system 3. (d) 4. (d)
5. It is dedicated since it is dedicated to doing one type of job.
6. Embedded system means the processor is embedded into that application.
7. Having multiple sources for a given part means you are not hostage to one supp
   importantly competition among suppliers brings about lower cost for that produc

SECTION 1.2: OVERVIEW OF THE 8051 FAMILY

1. 128 bytes of RAM, 4K bytes of on-chip ROM, four 8-bit I/O ports.
2. The 8052 has everything that the 8051 has, plus an extra timer, and the on-chip R
   bytes instead of 4K bytes. The RAM in the 8052 is 256 bytes instead of 128 byte
3. Both the 8051 and the 8031 have 128 bytes of RAM and the 8052 has 256 bytes.
4. (a) 4K bytes (b) 8K bytes (c) 0K bytes
5. 8
6. The difference is the type of on-chip ROM. In the 8751 it is UV-EPROM; in the A
   is flash; and in the DS5000 it is NV-RAM.
7. DS5000T has a real-time clock (RTC).
8. True
9. 12
10. 32

CHAPTER 2

8051 ASSEMBLY LANGUAGE PROGRAMMING

OBJECTIVES

Upon completion of this chapter, you will be able to:

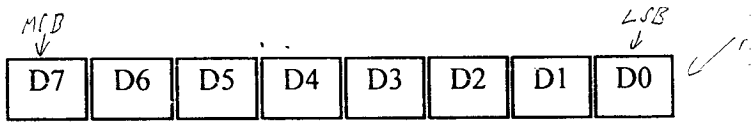
- **List the registers of the 8051 microcontroller**
- **Manipulate data using the registers and MOV instructions**
- **Code simple 8051 Assembly language instructions**
- **Assemble and run an 8051 program**
- **Describe the sequence of events that occur upon 8051 power-up**
- **Examine programs in ROM code of the 8051**
- **Explain the ROM memory map of the 8051**
- **Detail the execution of 8051 Assembly language instructions**
- **Describe 8051 data types**
- **Explain the purpose of the PSW (program status word) register**
- **Discuss RAM memory space allocation in the 8051**
- **Diagram the use of the stack in the 8051**
- **Manipulate the register banks of the 8051**

In Section 2.1 we look at the inside of the 8051. We demonstrate some of the widely used registers of the 8051 with simple instructions such as MOV and ADD. In Section 2.2 we examine Assembly language and machine language programming and define terms such as mnemonics, opcode, operand, etc. The process of assembling and creating a ready-to-run program for the 8051 is discussed in Section 2.3. Step-by-step execution of an 8051 program and the role of the program counter are examined in Section 2.4. In Section 2.5 we look at some widely used Assembly language directives, pseudocode, and data types related to the 8051. In Section 2.6 we discuss the flag bits and how they are affected by arithmetic instructions. Allocation of RAM memory inside the 8051 plus the stack and register banks of the 8051 are discussed in Section 2.7.

SECTION 2.1: INSIDE THE 8051

In this section we examine the major registers of the 8051 and show their use with the simple instructions MOV and ADD.

Registers



In the CPU, registers are used to store information temporarily. That information could be a byte of data to be processed, or an address pointing to the data to be fetched. The vast majority of 8051 registers are 8-bit registers. In the 8051 there is only one data type: 8 bits. The 8 bits of a register are shown in the diagram from the MSB (most significant bit) D7 to the LSB (least significant bit) D0. With an 8-bit data type, any data larger than 8 bits must be broken into 8-bit chunks before it is processed. Since there are a large number of registers in the 8051, we will concentrate on some of the widely used general-purpose registers and cover special registers in future chapters. See Appendix A.3 for a complete list of 8051 registers.

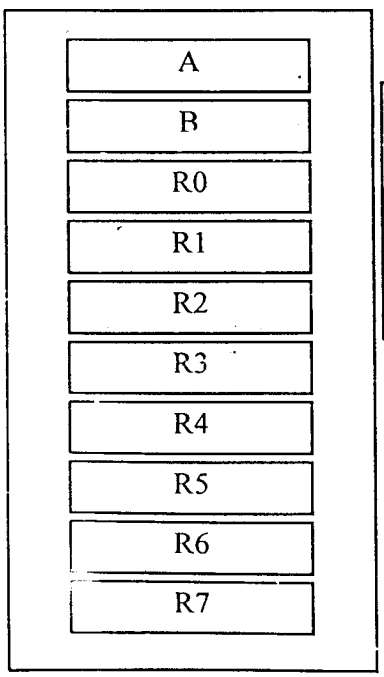


Figure 2-1 (a): Some 8-bit Registers of the 8051

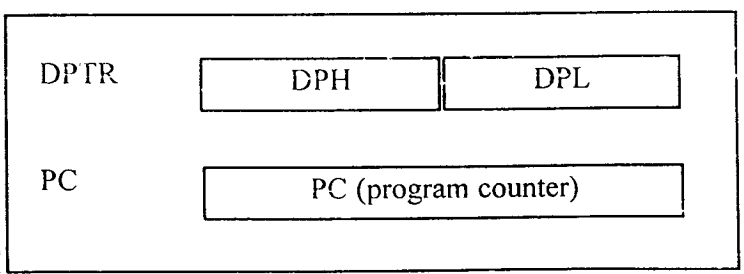


Figure 2-1 (b): Some 8051 16-bit Registers

The most widely used registers of the 8051 are A (accumulator), B, R0, R1, R2, R3, R4, R5, R6, R7, DPTR (data pointer), and PC (program counter). All of the above registers are 8-bits, except DPTR and the program counter. The accumulator, register A, is used for all arithmetic and logic instructions. To understand the use of these registers, we will show them in the context of two simple instructions, MOV and ADD.

MOV instruction

Simply stated, the MOV instruction copies data from one location to another. It has the following format:

```
MOV destination,source ;copy source to dest.
```

This instruction tells the CPU to move (in reality, copy) the source operand to the destination operand. For example, the instruction "MOV A,R0" copies the contents of register R0 to register A. After this instruction is executed, register A will have the same value as register R0. The MOV instruction does not affect the source operand. The following program first loads register A with value 55H (that is 55 in hex), then moves this value around to various registers inside the CPU. Notice the "#" in the instruction. This signifies that it is a value. The importance of this will be discussed soon.

```
MOV A,#55H      ;load value 55H into reg. A
MOV R0,A        ;copy contents of A into R0
                ;(now A=R0=55H)
MOV R1,A        ;copy contents of A into R1
                ;(now A=R0=R1=55H)
MOV R2,A        ;copy contents of A into R2
                ;now A=R0=R1=R2=55H)
MOV R3,#95H     ;load value 95H into R3
                ;(now R3=95H)
MOV A,R3        ;copy contents of R3 into A
                ;now A=R3=95H)
```

When programming the 8051 microcontroller, the following points should be noted:

1. Values can be loaded directly into any of registers A, B, or R0 - R7. However, to indicate that it is an immediate value it must be preceded with a pound sign (#). This is shown next.

```
MOV A,#23H      ;load 23H into A (A=23H)
MOV R0,#12H     ;load 12H into R0 (R0=12H)
MOV R1,#1FH     ;load 1FH into R1 (R1=1FH)
MOV R2,#2BH     ;load 2BH into R2 (R2=2BH)
MOV B,#3CH      ;load 3CH into B (B=3CH)
MOV R7,#9DH     ;load 9DH into R7 (R7=9DH)
MOV R5,#0F9H    ;load F9H into R5 (R5=F9H)
MOV R6,#12      ;load 12 decimal (= 0CH)
                ;into reg. R6 (R6=0CH)
```

Notice in instruction "MOV R5,#0F9H" that there is a need for 0 between the # and F to indicate that F is a hex number and not a letter. In other words "MOV R5,#F9H" will cause an error.

2. If values 0 to F are moved into an 8-bit register, the rest of the bits are assumed to be all zeros. For example, in "MOV A, #5" the result will be A = 05; that is, A = 00000101 in binary.

3. Moving a value that is too large into a register will cause an error.

```
MOV A,#7F2H;ILLEGAL: 7F2H > 8 bits (FFH)
MOV R2,456 ;ILLEGAL: 456 > 255 decimal (FFH)
```

4. To load a value into a register it must be preceded with a pound sign (#). Otherwise it means to load from a memory location. For example "MOV A, 17H" means to move into A the value held in memory location 17H, which could have any value. In order to load the value 17H into the accumulator we must write "MOV A, #17H" with the # preceding the number. Notice that the absence of the pound sign will not cause an error by the assembler since it is a valid instruction. However, the result would not be what the programmer intended. This is a common error for beginning programmers in the 8051.

ADD instruction

The ADD instruction has the following format:

```
ADD A,source    ;ADD the source operand
                ;to the accumulator
```

The ADD instruction tells the CPU to add the source byte to register A and put the result in register A. To add two numbers such as 25H and 34H, each can be moved to a register and then added together:

```
MOV A,#25H      ;load 25H into A
MOV R2,#34H     ;load 34H into R2
ADD A,R2        ;add R2 to accumulator
                ;(A = A + R2)
```

Executing the program above results in A = 59H (25H + 34H = 59H) and R2 = 34H. Notice that the content of R2 does not change. The program above can be written in many ways, depending on the registers used. Another way might be:

```
MOV R5,#25H     ;load 25H into R5 (R5=25H)
MOV R7,#34H     ;load 34H into R7  (R7=34H)
MOV A,#0        ;load 0 into A (A=0,clear A)
ADD A,R5        ;add to A content of R5
                ;where A = A + R5
ADD A,R7        ;add to A content of R7
                ;where A = A + R7
```

The program above results in A = 59H. There are always many ways to write the same program. One question that might come to mind after looking at the program above, is whether it is necessary to move both data items into registers

before adding them together. The answer is no, it is not necessary. Look at the following variation of the same program:

```
MOV A,#25H ;load one operand into A (A=25H)
ADD A,#34H ;add the second operand 34H to A
```

In the above case, while one register contained one value, the second value followed the instruction as an operand. This is called an *immediate* operand. The examples shown so far for the ADD instruction indicate that the source operand can be either a register or immediate data, but the destination must always be register A, the accumulator. In other words, an instruction such as "ADD R2,#12H" is invalid since register A (accumulator) must be involved in any arithmetic operation. Notice that "ADD R4,A" is also invalid for the reason that A must be the destination of any arithmetic operation. To put it simply: In the 8051, register A must be involved and be the destination for all arithmetic operations. The foregoing discussion explains the reason that register A is referred to as the accumulator. The format for Assembly language instructions, descriptions of their use, and a listing of legal operand types are provided in Appendix A.1.

There are two 16-bit registers in the 8051: PC (program counter) and DPTR (data pointer). The importance and use of the program counter are covered in Section 2.3. The DPTR register is used in accessing data and is discussed in Chapter 5 when addressing modes are covered.

Review Questions

1. Write the instructions to move value 34H into register A and value 3FH into register B, then add them together.
2. Write the instructions to add the values 16H and CDH. Place the result in register R2.
3. True or false. No value can be moved directly into registers R0 - R7.
4. What is the largest hex value that can be moved into an 8-bit register? What is the decimal equivalent of the hex value?
5. The vast majority of registers in 8051 are _____ bits.

SECTION 2.2: INTRODUCTION TO 8051 ASSEMBLY PROGRAMMING

In this section we discuss Assembly language format and define some widely used terminology associated with Assembly language programming.

While the CPU can work only in binary, it can do so at a very high speed. However, it is quite tedious and slow for humans to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called *machine language*. In the early days of the computer, programmers coded programs in machine language. Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, Assembly languages were developed which provided mnemonics for the machine code instructions, plus other features which made programming faster and less prone to error. The term *mnemonic* is frequently used in computer science and engineering literature to refer to codes and abbre-

viations that are relatively easy to remember. Assembly language programs must be translated into machine code by a program called an *assembler*. Assembly language is referred to as a *low-level language* because it deals directly with the internal structure of the CPU. To program in Assembly language, the programmer must know all the registers of the CPU and the size of each, as well as other details.

Today, one can use many different programming languages, such as BASIC, Pascal, C, C++, Java, and numerous others. These languages are called *high-level languages* because the programmer does not have to be concerned with the internal details of the CPU. Whereas an *assembler* is used to translate an Assembly language program into machine code (sometimes also called *object code* or opcode for operation code), high-level languages are translated into machine code by a program called a *compiler*. For instance, to write a program in C, one must use a C compiler to translate the program into machine language. Now we look at 8051 Assembly language format and use an 8051 assembler to create a ready-to-run program.

Structure of Assembly language

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items.

```
      ORG   0H          ;start (origin) at location 0
      MOV   R5,#25H     ;load 25H into R5
      MOV   R7,#34H     ;load 34H into R7
      MOV   A,#0        ;load 0 into A
      ADD   A,R5        ;add contents of R5 to A
                        ;now A = A + R5
      ADD   A,R7        ;add contents of R7 to A
                        ;now A = A + R7
      ADD   A,#12H      ;add to A value 12H
                        ;now A = A + 12H
HERE: SJMP  HERE        ;stay in this loop
      END               ;end of asm source file
```

Program 2-1: Sample of an Assembly Language Program

A given Assembly language program (see Program 2-1) is a series of statements, or lines, which are either Assembly language instructions such as ADD and MOV, or statements called directives. While instructions tell the CPU what to do, directives (also called pseudo-instructions) give directions to the assembler. For example, in the above program while the MOV and ADD instructions are commands to the CPU, ORG and END are directives to the assembler. ORG tells the

assembler to place the opcode at memory location 0 while END indicates to the assembler the end of the source code. In other words, one is for the start of the program and the other one for the end of the program.

An Assembly language instruction consists of four fields:

```
[label:]    mnemonic   [operands]   [;comment]
```

Brackets indicate that a field is optional and not all lines have them. Brackets should not be typed in. Regarding the above format, the following points should be noted.

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed a certain number of characters. Check your assembler for the rule.

2. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as

```
ADD A,B
MOV A,#67
```

   ADD and MOV are the mnemonics which produce opcodes; "A,B" and "A,#67" are the operands. Instead of a mnemonic and operand, these two fields could contain assembler pseudo-instructions, or directives. Remember that directives do not generate any machine code (opcode) and are used only by the assembler, as opposed to instructions that are translated into machine code (opcode) for the CPU to execute. In Program 2-1 the commands ORG (origin) and END are examples of directives (some 8051 assemblers use .ORG and .END). Check your assembler for the rules. More of these pseudo-instructions are discussed in detail in Section 2.5.

3. The comment field begins with a semicolon comment indicator ";". Comments may be at the end of a line or on a line by themselves. The assembler ignores comments, but they are indispensable to programmers. Although comments are optional, it is recommended that they be used to describe the program in order to make it easier for someone else to read and understand.

4. Notice the label "HERE" in the label field in Program 2-1. Any label referring to an instruction must be followed by a colon symbol, ":". In the SJMP (short jump instruction), the 8051 is told to stay in this loop indefinitely. If your system has a monitor program you do not need this line and it should be deleted from your program. In the next section we will see how to create a ready-to-run program.

Review Questions

1. What is the purpose of pseudo-instructions?
2. _____ are translated by the assembler into machine code, whereas _____ are not.
3. True or false. Assembly language is a high-level language.
4. Which of the following produces opcode?
   (a) ADD A,R2 (b) MOV A,#12 (c) ORG 2000H (d) SJMP HERE
5. Pseudo-instructions are also called _____.
6. True or false. Assembler directives are not used by the CPU itself. They are simply a guide to the assembler.
7. In question 4, which one is an assembler directive?

SECTION 2.3: ASSEMBLING AND RUNNING AN 8051 PROGRAM

Now that the basic form of an Assembly language program has been given, the next question is: How it is created, assembled and made ready to run? The steps to create an executable Assembly language program are outlined as follows.

1. First we use an editor to type in a program similar to Program 2-1. Many excellent editors or word processors are available that can be used to create and/or edit the program. A widely used editor is the MS-DOS EDIT program (or Notepad in Windows), which comes with all Microsoft operating systems. Notice that the editor must be able to produce an ASCII file. For many assemblers, the file names follow the usual DOS conventions, but the source file has the extension "asm" or "src", depending on which assembler you are using. Check your assembler for the convention. The "asm" extension for the source file is used by an assembler in the next step.

2. The "asm" source file containing the program code created in step 1 is fed to an 8051 assembler. The assembler converts the instructions into

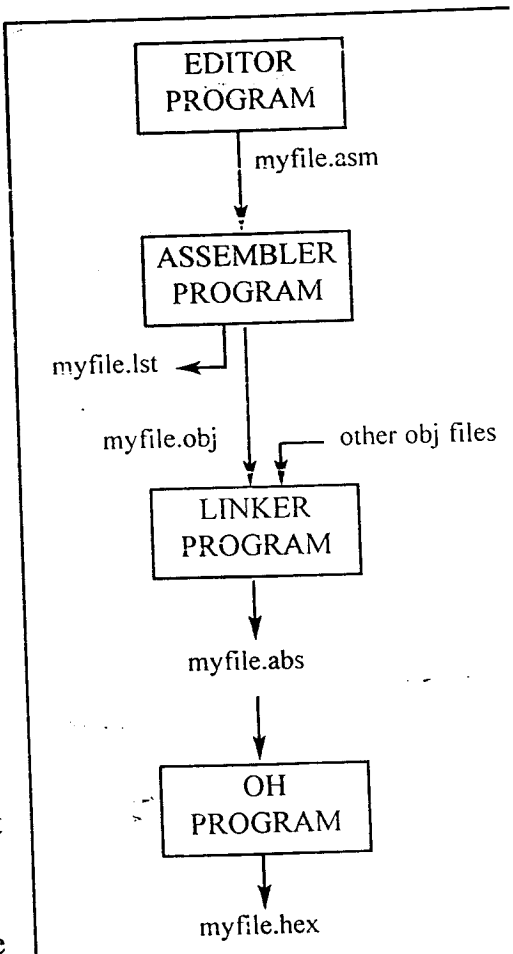


Figure 2-2. Steps to Create a Program

machine code. The assembler will produce an object file and a list file. The extension for the object file is "obj" while the extension for the list file is "lst".

3. Assemblers require a third step called *linking*. The link program takes one or more object files and produces an absolute object file with the extension "abs". This abs file is used by 8051 trainers that have a monitor program.

4. Next the "abs" file is fed into a program called "OH" (object to hex converter) which creates a file with extension "hex" that is ready to burn into ROM. This program comes with all 8051 assemblers. Recent Windows-based assemblers combine steps 2 through 4 into one step.

More about "asm" and "obj" files

The "asm" file is also called the *source* file and for this reason some assemblers require that this file have the "src" extension. Check your 8051 assembler to see which one it requires. As mentioned earlier, this file is created with an editor such as DOS EDIT or Window's Notepad. The 8051 assembler converts the asm file's Assembly language instructions into machine language and provides the obj (object) file. In addition to creating the object file, the assembler also produces the lst file (list file).

lst file

The lst (list) file, which is optional, is very useful to the programmer because it lists all the opcodes and addresses as well as errors that the assembler detected. Many assemblers assume that the list file is not wanted unless you indicate that you want to produce it. This file can be accessed by an editor such as DOS EDIT and displayed on the monitor or sent to the printer to get a hard copy. The programmer uses the list file to find syntax errors. It is only after fixing all the errors indicated in the lst file that the obj file is ready to be input to the linker program.

```
1 0000                ORG 0H          ;start (origin) at 0
2 0000 7D25           MOV R5,#25H     ;load 25H into R5
3 0002 7F34           MOV R7,#34H     ;load 34H into R7
4 0004 7400           MOV A,#0        ;load 0 into A
5 0006 2D             ADD A,R5        ;add contents of R5 to A
                                      ;now A = A + R5
6 0007 2F             ADD A,R7        ;add contents of R7 to A
                                      ;now A = A + R7
7 0008 2412           ADD A,#12H      ;add to A value 12H
                                      ;now A = A + 12H
8 000A 80FE HERE:     SJMP HERE       ;stay in this loop
  000C                END             ;end of asm source file
```

Program 2-1: List File

Review Questions

1. True or false. The DOS program EDIT produces an ASCII file.
2. True or false. Generally, the extension of the source file is ".asm" or ".src".
3. Which of the following files can be produced by the DOS EDIT program?
   (a) myprog.asm (b) myprog.obj (c) myprog.exe (d) myprog.lst
4. Which of the following files is produced by an 8051 assembler?
   (a) myprog.asm (b) myprog.obj (c) myprog.hex (d) myprog.lst
5. Which of the following files lists syntax errors?
   (a) myprog.asm (b) myprog.obj (c) myprog.hex (d) myprog.lst

SECTION 2.4: THE PROGRAM COUNTER AND ROM SPACE IN THE 8051

In this section we examine the role of the program counter (PC) register in executing an 8051 program. We also discuss ROM memory space for various 8051 family members.

Program counter in the 8051

Another important register in the 8051 is the PC (program counter). The program counter points to the address of the next instruction to be executed. As the CPU fetches the opcode from the program ROM, the program counter is incremented to point to the next instruction. The program counter in the 8051 is 16 bits wide. This means that the 8051 can access program addresses 0000 to FFFFH, a total of 64K bytes of code. However, not all members of the 8051 have the entire 64K bytes of on-chip ROM installed, as we will see soon. Where does the 8051 wake up when it is powered? We will discuss this important topic next.

Where the 8051 wakes up when it is powered up

One question that we must ask about any microcontroller (or microprocessor) is: At what address does the CPU wake up upon applying power to it? Each microprocessor is different. In the case of the 8051 family, that is, all members regardless of the maker and variation, the microcontroller wakes up at memory address 0000 when it is powered up. By powering up we mean applying V_{CC} to the RESET pin as discussed in Chapter 4. In other words, when the 8051 is powered up, the PC (program counter) has the value of 0000 in it. This means that it expects the first opcode to be stored at ROM address 0000H. For this reason in the 8051 system, the first opcode must be burned into memory location 0000H of program ROM since this is where it looks for the first instruction when it is booted. We achieve this by the ORG statement in the source program as shown earlier. Next we discuss the step-by-step action of the program counter in fetching and executing a sample program.

Placing code in program ROM

To get a better understanding of the role of the program counter in fetching and executing a program, we examine the action of the program counter as each instruction is fetched and executed. First, we examine once more the list file

of the sample program and how the code is placed in the ROM of an 8051 chip. As we can see, the opcode and operand for each instruction are listed on the left side of the list file.

```
0000                    ORG 0H          ;start at location 0
0000 7D25               MOV R5,#25H     ;load 25H into R5
0002 7F34               MOV R7,#34H     ;load 34H into R7
0004 7400               MOV A,#0        ;load 0 into A
0006 2D                 ADD A,R5        ;add contents of R5 to A
                                        ;now A = A + R5
0007 2F                 ADD A,R7        ;add contents of R7 to A
                                        ;now A = A + R7
0008 2412               ADD A,#12H      ;add to A value 12H
                                        ;now A = A + 12H
000A 80FE HERE:         SJMP HERE       ;stay in this loop
000C                    END             ;end of asm source file
```

Program 2-1: List File

ROM Address	Machine Language	Assembly Language
0000	7D25	MOV R5,#25H
0002	7F34	MOV R7,#34H
0004	7400	MOV A,#0
0006	2D	ADD A,R5
0007	2F	ADD A,R7
0008	2412	ADD A,#12H
000A	80FE	HERE: SJMP HERE

After the program is burned into ROM of an 8051 family member such as 8751 or AT8951 or DS5000, the opcode and operand are placed in ROM memory locations starting at 0000 as shown in the list below.

The list shows that address 0000 contains 7D which is the opcode for moving a value into register R5, and address 0001 contains the operand (in this case 25H) to be moved to R5. Therefore, the instruction "MOV R5,#25H" has a machine code of "7D25", where 7D is the opcode and 25 is the operand. Similarly, the machine code "7F34" is located in memory locations 0002 and 0003 and represents the opcode and the operand for the instruction "MOV R7,#34H". In the same way, machine code "7400" is located in memory locations 0004 and 0005 and represents the opcode and the operand for the instruction "MOV A,#0". The memory location 0006 has the opcode of 2D which is the opcode for the

Program 2-1: ROM Contents

Address	Code
0000	7D
0001	25
0002	7F
0003	34
0004	74
0005	00
0006	2D
0007	2F
0008	24
0009	12
000A	80
000B	FE

instruction "ADD A,R5" and memory location 0007 has the content 2F, which is opcode for the "ADD A,R7" instruction. The opcode for instruction "ADD A,#12H" is located at address 0008 and the operand 12H at address 0009. The memory location 000A has the opcode for the SJMP instruction and its target address is located in location 000B. The reason the target address is FE is explained in the next chapter.

Executing a program byte by byte

Assuming that the above program is burned into the ROM of an 8051 chip (or 8751, AT8951, or DS5000), the following is a step-by-step description of the action of the 8051 upon applying power to it.

1. When the 8051 is powered up, the PC (program counter) has 0000 and starts to fetch the first opcode from location 0000 of the program ROM. In the case of the above program the first opcode is 7D, which is the code for moving an operand to R5. Upon executing the opcode, the CPU fetches the value 25 and places it in R5. Now one instruction is finished. Then the program counter is incremented to point to 0002 (PC = 0002), which contains opcode 7F, the opcode for the instruction "MOV R7, . .".
2. Upon executing the opcode 7F, the value 34H is moved into R7. Then the program counter is incremented to 0004.
3. ROM location 0004 has the opcode for instruction "MOV A,#0". This instruction is executed and now PC=0006. Notice that all the above instructions are 2-byte instructions; that is, each one takes two memory locations.
4. Now PC = 0006 points to the next instruction which is "ADD A,R5". This is a 1-byte instruction. After the execution of this instruction, PC = 0007.
5. The location 0007 has the opcode 2F which belongs to the instruction "ADD A,R7". This is also a 1-byte instruction. Upon execution of this instruction, PC is incremented to 0008. This process goes on until all the instructions are fetched and executed. The fact the program counter points at the next instruction to be executed explains why some microprocessors (notably the x86) call the program counter the *instruction pointer*.

ROM memory map in the 8051 family

As we saw in the last chapter, some family members have only 4K bytes of on-chip ROM (e.g., 8751, AT8951) and some, such as the AT89C52, have 8K bytes of ROM. Dallas Semiconductor's DS5000-32 has 32K bytes of on-chip ROM. Dallas Semiconductor also has an 8051 with 64K bytes of on-chip ROM. The point to remember is that no member of the 8051 family can access more than 64K bytes of opcode since the program counter in the 8051 is a 16-bit register (0000 to FFFF address range). It must be noted that while the first location of program ROM inside the 8051 has the address of 0000, the last location can be different depending on the size of the ROM on the chip. Among the 8051 family members, the 8751 and AT8951 have 4K bytes of on-chip ROM. This 4K bytes ROM memory has memory addresses of 0000 to 0FFFH. Therefore, the first location of on-chip ROM of this 8051 has an address of 0000 and the last location has the address of 0FFFH. Look at Example 2-1 to see how this is computed.

Example 2-1

Find the ROM memory address of each of the following 8051 chips.
(a) AT89C51 (or 8751) with 4KB (b) DS5000-32 with 32KB

Solution:

(a) With 4K bytes of on-chip ROM memory space, we have 4096 bytes, which is 1000H in hex (4 × 1024 = 4096 or 1000 in hex). This much memory maps to address locations of 0000 to 0FFFH. Notice that 0 is always the first location.

(b) With 32K bytes we have 32,768 (32 × 1024 = 32,768) bytes. Converting 32,768 to hex, we get 8000H; therefore, the memory space is 0000 to 7FFFH.

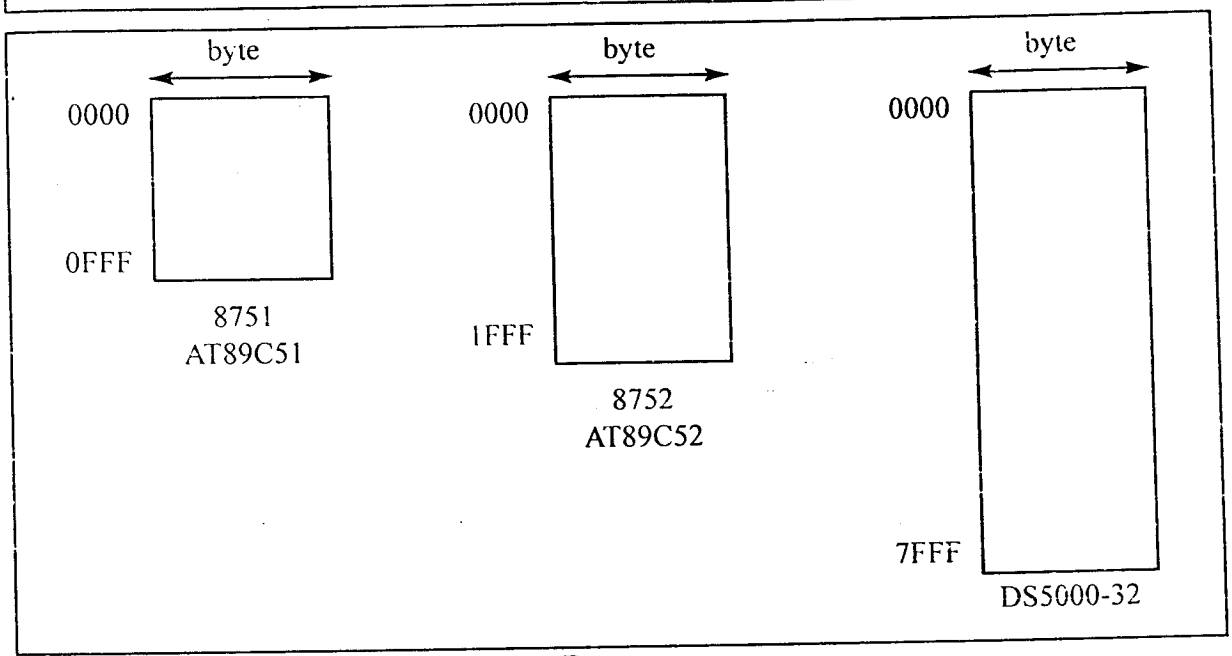


Figure 2-3. 8051 On-Chip ROM Address Range

Review Questions

1. In the 8051, the program counter is _____ bits wide.
2. True or false. Every member of the 8051 family, regardless of the maker, wakes up at memory 0000H when it is powered up.
3. At what ROM location do we store the first opcode of an 8051 program?
4. The instruction "`MOV A,#44H`" is a ____-byte instruction.
5. What is the ROM address space for the 8052 chip?

SECTION 2.5: 8051 DATA TYPES AND DIRECTIVES

In this section we look at some widely used data types and directives supported by the 8051 assembler.

8051 data type and directives

The 8051 microcontroller has only one data type. It is 8 bits, and the size of each register is also 8 bits. It is the job of the programmer to break down data

larger than 8 bits (00 to FFH, or 0 to 255 in decimal) to be processed by the CPU. For examples of how to process data larger than 8 bits, see Chapter 6. The data types used by the 8051 can be positive or negative. A discussion of signed numbers is given in Chapter 6.

DB (define byte)

The DB directive is the most widely used data directive in the assembler. It is used to define the 8-bit data. When DB is used to define data, the numbers can be in decimal, binary, hex, or ASCII formats. For decimal, the "D" after the decimal number is optional, but using "B" (binary) and "H" (hexadecimal) for the others is required. Regardless of which is used, the assembler will convert the numbers into hex. To indicate ASCII, simply place it in quotation marks ('like this'). The assembler will assign the ASCII code for the numbers or characters automatically. The DB directive is the only directive that can be used to define ASCII strings larger than two characters; therefore, it should be used for all ASCII data definitions. Following are some DB examples:

```
        ORG   500H
DATA1:  DB    28              ;DECIMAL(1C in hex)
DATA2:  DB    00110101B       ;BINARY (35 in hex)
DATA3:  DB    39H             ;HEX
        ORG   510H
DATA4:  DB    "2591"          ;ASCII NUMBERS
        ORG   518H
DATA6:  DB    "My name is Joe";ASCII CHARACTERS
```

Either single or double quotes can be used around ASCII strings. This can be useful for strings, which contain a single quote such as "O'Leary". DB is also used to allocate memory in byte-sized chunks.

Assembler directives

The following are some more widely used directives of the 8051.

ORG (origin)

The ORG directive is used to indicate the beginning of the address. The number that comes after ORG can be either in hex or in decimal. If the number is not followed by H, it is decimal and the assembler will convert it to hex. Some assemblers use ". ORG" (notice the dot) instead of "ORG" for the origin directive. Check your assembler.

EQU (equate)

This is used to define a constant without occupying a memory location. The EQU directive does not set aside storage for a data item but associates a constant value with a data label so that when the label appears in the program, its constant value will be substituted for the label. The following uses EQU for the counter constant and then the constant is used to load the R3 register.

```
COUNT       EQU  25
...         ....
MOV         R3,#COUNT
```

When executing the instruction "MOV R3,#COUNT", the register R3 will be loaded with the value 25 (notice the # sign). What is the advantage of using EQU? Assume that there is a constant (a fixed value) used in many different places in the program, and the programmer wants to change its value throughout. By the use of EQU, one can change it once and the assembler will change all of its occurrences, rather than search the entire program trying to find every occurrence.

END directive

Another important pseudocode is the END directive. This indicates to the assembler the end of the source (asm) file. The END directive is the last line of an 8051 program, meaning that in the source code anything after the END directive is ignored by the assembler. Some assemblers use ". END" (notice the dot) instead of "END".

Rules for labels in Assembly language

By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow. First, each label name must be unique. The names used for labels in Assembly language programming consist of alphabetic letters in both upper and lower case, the digits 0 through 9, and the special characters question mark (?), period (.), at (@), underline (_), and dollar sign ($). The first character of the label must be an alphabetic character. In other words it cannot be a number. Every assembler has some reserved words which must not be used as labels in the program. Foremost among the reserved words are the mnemonics for the instructions. For example, "MOV" and "ADD" are reserved since they are instruction mnemonics. Aside from the mnemonics there are some other reserved words. Check your assembler for the list of reserved words.

Review Questions

1. The _____ directive is always used for ASCII strings.
2. How many bytes are used by the following?
   `DATA_1 DB "AMERICA"`
3. What is the advantage in using the EQU directive to define a constant value?
4. How many bytes are set aside by each of the following directives?
   (a) `ASC_DATA DB "1234"` (b) `MY_DATA DB "ABC1234"`
5. State the contents of memory locations 200H - 205H for the following

```
            ORG   200H
MYDATA:     DB    "ABC123"
```

SECTION 2.6: 8051 FLAG BITS AND THE PSW REGISTER

Like any other microprocessor, the 8051 has a flag register to indicate arithmetic conditions such as the carry bit. The flag register in the 8051 is called the program status word (PSW) register. In this section we discuss various bits of this register and provide some examples of how it is altered.

PSW (program status word) register

The program status word (PSW) register is an 8-bit register. It is also referred to as the *flag register*. Although the PSW register is 8 bits wide, only 6 bits of it are used by the 8051. The two unused bits are user-definable flags. Four of the flags are called conditional flags, meaning that they indicate some conditions that resulted after an instruction was executed. These four are CY (carry), AC (auxiliary carry), P (parity), and OV (overflow).

As seen from Figure 2-4, the bits PSW3 and PSW4 are designated as RS0 and RS1, and are used to change the bank registers. They are explained in the next section. The PSW.5 and PSW.1 bits are general-purpose status flag bits and can be used by the programmer for any purpose. In other words, they are user definable. See Figure 2-4 for the bits of the PSW register.

CY	AC	F0	RS1	RS0	OV	--	P

CY	PSW.7	Carry flag.
AC	PSW.6	Auxiliary carry flag.
--	PSW.5	Available to the user for general purpose.
RS1	PSW.4	Register Bank selector bit 1.
RS0	PSW.3	Register Bank selector bit 0.
OV	PSW.2	Overflow flag.
--	PSW.1	User definable bit.
P	PSW.0	Parity flag. Set/cleared by hardware each instuction cycle to indicate an odd/even number of 1 bits in the accumulator.

RS1	RS0	Register Bank	Address
0	0	0	00H - 07H
0	1	1	08H - 0FH
1	0	2	10H - 17H
1	1	3	18H - 1FH

Figure 2-4. Bits of the PSW Register

The following is a brief explanation of four of the flag bits of the PSW register. The impact of instructions on these registers is then discussed.

CY, the carry flag

This flag is set whenever there is a carry out from the d7 bit. This flag bit is affected after an 8-bit addition or subtraction. It can also be set to 1 or 0 directly by an instruction such as "SETB C" and "CLR C" where "SETB C" stands for "set bit carry" and "CLR C" for "clear carry". More about these and other bit-addressable instructions will be given in Chapter 8.

AC, the auxiliary carry flag

If there is a carry from D3 to D4 during an ADD or SUB operation, this bit is set; otherwise, it is cleared. This flag is used by instructions that perform BCD (binary coded decimal) arithmetic. See Chapter 6 for more information.

P, the parity flag

The parity flag reflects the number of 1s in the A (accumulator) register only. If the A register contains an odd number of 1s, then P = 1. Therefore, P = 0 if A has an even number of 1s.

OV, the overflow flag

This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations. The overflow flag is only used to detect errors in signed arithmetic operations and is discussed in detail in Chapter 6.

ADD instruction and PSW

Next we examine the impact of the ADD instruction on the flag bits CY, AC, and P of the PSW register. Some examples should clarify their status. Although the flag bits affected by the ADD instruction are CY (carry flag), P (parity flag), AC (auxiliary carry flag), and OV (overflow flag) we will focus on flags CY, AC, and P for now. A discussion of the overflow flag is given in Chapter 6, since it relates only to signed number arithmetic. How the various flag bits are used in programming is discussed in future chapters in the context of many applications.

See Examples 2-2 through 2-4 for the impact on selected flag bits as a result of the ADD instruction.

Table 2-1: Instructions That Affect Flag Bits

Instruction	CY	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,bit	X		
ANL C,/bit	X		
ORL C,bit	X		
ORL C,/bit	X		
MOV C,bit	X		
CJNE	X		

Note: X can be 0 or 1.

Example 2-2

Show the status of the CY, AC, and P flags after the addition of 38H and 2FH in the following instructions.

```
MOV A,#38H
ADD A,#2FH ;after the addition A=67H, CY=0
```

Solution:

```
  38      00111000
+ 2F      00101111
  67      01100111
```

CY = 0 since there is no carry beyond the D7 bit
AC = 1 since there is a carry from the D3 to the D4 bit
P = 1 since the accumulator has an odd number of 1s (it has five 1s).

Example 2-3

Show the status of the CY, AC, and P flags after the addition of 9CH and 64H in the following instructions.

```
MOV A,#9CH
ADD A,#64H      ;after addition A=00 and CY=1
```

Solution:

```
  9C      10011100
+ 64      01100100
 100      00000000
```

CY=1 since there is a carry beyond the D7 bit
AC=1 since there is a carry from the D3 to the D4 bit
P=0 since the accumulator has an even number of 1s (it has zero 1s).

Example 2-4

Show the status of the CY, AC, and P flags after the addition of 88H and 93H in the following instructions.

```
MOV A,#88H
ADD A,#93H      ;after the addition A=1BH,CY=1
```

Solution:

```
  88      10001000
+ 93      10010011
 11B      00011011
```

CY=1 since there is a carry beyond the D7 bit.
AC=0 since there is no carry from the D3 to the D4 bit.
P=0 since the accumulator has an even number of 1s (it has four 1s).

Review Questions

1. The flag register in the 8051 is called _____.
2. What is the size of the flag register in the 8051?
3. Which bits of the PSW register are user-definable?
4. Find the CY and AC flag bits for the following code.

```
MOV   A,#0FFH
ADD   A,#01
```

5. Find the CY and AC flag bits for the following code.

```
MOV   A,#0C2H
ADD   A,#3DH
```

SECTION 2.7: 8051 REGISTER BANKS AND STACK

The 8051 microcontroller has a total of 128 bytes of RAM. In this section we discuss the allocation of these 128 bytes of RAM and examine their usage as registers and stack.

RAM memory space allocation in the 8051

There are 128 bytes of RAM in the 8051 (Some members, notably the 8052, have 256 bytes of RAM). The 128 bytes of RAM inside the 8051 are assigned addresses 00 to 7FH. As we will see in Chapter 5, they can be accessed directly as memory locations. These 128 bytes are divided into three different groups as follows.

1. A total of 32 bytes from locations 00 to 1F hex are set aside for register banks and the stack.
2. A total of 16 bytes from locations 20H to 2FH are set aside for bit-addressable read/write memory. A detailed discussion of bit-addressable memory and instructions is given in Chapter 8.
3. A total of 80 bytes from locations 30H to 7FH are used for read and write storage, or what is normally called a *scratch pad*. These 80 locations of RAM are widely used for the purpose of storing data and parameters by 8051 programmers. We will use them in future chapters to store data brought into the CPU via I/O ports.

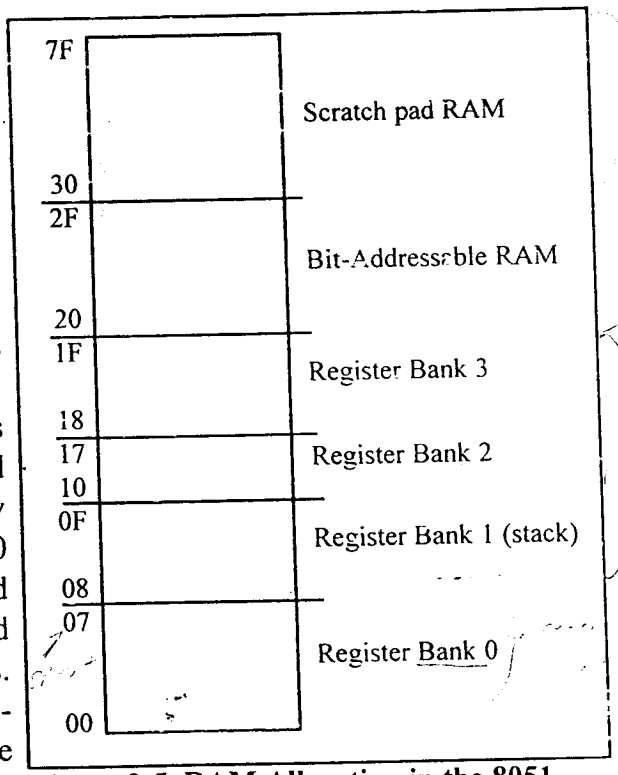


Figure 2-5. RAM Allocation in the 8051

Register banks in the 8051

As mentioned earlier, a total of 32 bytes of RAM are set aside for the register banks and stack. These 32 bytes are divided into 4 banks of registers in which

each bank has 8 registers, R0 - R7. RAM locations from 0 to 7 are set aside for bank 0 of R0 - R7 where R0 is RAM location 0, R1 is RAM location 1, R2 is location 2, and so on, until memory location 7 which belongs to R7 of bank 0. The second bank of registers R0 - R7 starts at RAM location 08 and goes to location 0FH. The third bank of R0 - R7 starts at memory location 10H and goes to location 17H; finally, RAM locations 18H to 1FH are set aside for the fourth bank of R0 - R7. The following shows how the 32 bytes are allocated into 4 banks:

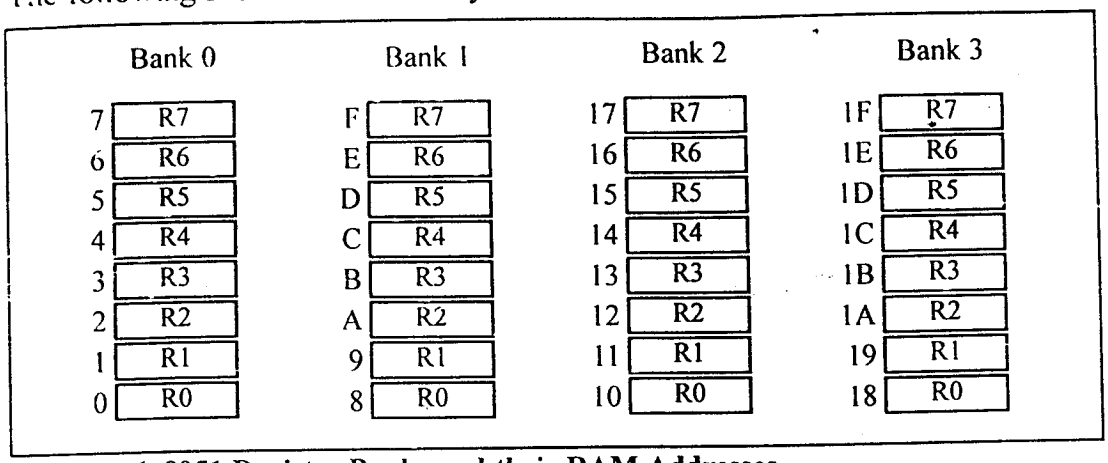


Figure 2-6. 8051 Register Banks and their RAM Addresses

As we can see from Figure 2-5, bank 1 uses the same RAM space as the stack. This is a major problem in programming the 8051. We must either not use register bank 1, or we must allocate another area of RAM for the stack. This will be discussed below.

Example 2-5

State the contents of RAM locations after the following program:

```
MOV R0,#99H    ;load R0 with value 99H
MOV R1,#85H    ;load R1 with value 85H
MOV R2,#3FH    ;load R2 with value 3FH
MOV R7,#63H    ;load R7 with value 63H
MOV R5,#12H    ;load R5 with value 12H
```

Solution:

After the execution of the above program we have the following:
RAM location 0 has value 99H RAM location 1 has value 85H
RAM location 2 has value 3FH RAM location 7 has value 63H
RAM location 5 has value 12H

Default register bank

If RAM locations 00 - 1F are set aside for the four register banks, which register bank of R0 - R7 do we have access to when the 8051 is powered up? The answer is register bank 0; that is, RAM locations 0, 1, 2, 3, 4, 5, 6, and 7 are accessed with the names R0, R1, R2, R3, R4, R5, R6, and R7 when programming the 8051. It is much easier to refer to these RAM locations with names such as R0, R1, and so on, than by their memory locations. Example 2-6 clarifies this concept.

Example 2-6

Repeat Example 2-5 using RAM addresses instead of register names.

Solution:

This is called direct addressing mode and uses the RAM address location for the destination address. See Chapter 5 for a more detailed discussion of addressing modes.

```
MOV 00,#99H     ;load R0 with value 99H
MOV 01,#85H     ;load R1 with value 85H
MOV 02,#3FH     ;load R2 with value 3FH
MOV 07,#63H     ;load R7 with value 63H
MOV 05,#12H     ;load R5 with value 12H
```

How to switch register banks

As stated above, register bank 0 is the default when the 8051 is powered up. We can switch to other banks by use of the PSW (program status word) register. Bits D4 and D3 of the PSW are used to select the desired register bank as shown in Table 2-2.

Table 2.2: PSW Bits Bank Selection

	RS1 (PSW.4)	RS0 (PSW.3)
Bank 0	0	0
Bank 1	0	1
Bank 2	1	0
Bank 3	1	1

The D3 and D4 bits of register PSW are often referred to as PSW.4 and PSW.3 since they can be accessed by the bit-addressable instructions SETB and CLR. For example, "SETB PSW.3" will make PSW.3=1 and select bank register 1. See Example 2-7.

Example 2-7

State the contents of the RAM locations after the following program:

```
SETB PSW.4      ;select bank 2
MOV R0,#99H     ;load R0 with value 99H
MOV R1,#85H     ;load R1 with value 85H
MOV R2,#3FH     ;load R2 with value 3FH
MOV R7,#63H     ;load R7 with value 63H
MOV R5,#12H     ;load R5 with value 12H
```

Solution:

By default, PSW.3=0 and PSW.4=0; therefore, the instruction "SETB PSW.4" sets RS1=1 and RS0=0, thereby selecting register bank 2. Register bank 2 uses RAM locations 10H - 17H. After the execution of the above program we have the following:

RAM location 10H has value 99H RAM location 11H has value 85H
RAM location 12H has value 3FH RAM location 17H has value 63H
RAM location 15H has value 12H

Stack in the 8051

The stack is a section of RAM used by the CPU to store information temporarily. This information could be data or an address. The CPU needs this storage area since there are only a limited number of registers.

How stacks are accessed in the 8051

If the stack is a section of RAM, there must be registers inside the CPU to point to it. The register used to access the stack is called the SP (stack pointer) register. The stack pointer in the 8051 is only 8 bits wide, which means that it can take values of 00 to FFH. When the 8051 is powered up, the SP register contains value 07. This means that RAM location 08 is the first location being used for the stack by the 8051. The storing of a CPU register in the stack is called a PUSH, and loading the contents of the stack back into a CPU register is called a POP. In other words, a register is pushed onto the stack to save it and popped off the stack to retrieve it. The job of the SP is very critical when push and pop actions are performed. To see how the stack works, let's look at the PUSH and POP instructions.

Pushing onto the stack

In the 8051 the stack pointer (SP) is pointing to the last used location of the stack. As we push data onto the stack, the stack pointer (SP) is incremented by one. Notice that this is different from many microprocessors, notably x86 processors in which the SP is decremented when data is pushed onto the stack. Examining Example 2-8, we see that as each PUSH is executed, the contents of the register are saved on the stack and SP is incremented by 1. Notice that for every byte of data saved on the stack, SP is incremented only once. Notice also that to push the registers onto the stack we must use their RAM addresses. For example, the instruction "`PUSH 1`" pushes register R1 onto the stack.

Example 2-8

Show the stack and stack pointer for the following. Assume the default stack are

```
MOV  R6,#25H
MOV  R1,#12H
MOV  R4,#0F3H
PUSH 6
PUSH 1
PUSH 4
```

Solution:

	After PUSH 6	After PUSH 1	After PUSH 4
0B	0B	0B	0B
0A	0A	0A	0A F3
09	09	09 12	09 12
08	08 25	08 25	08 25
Start SP = 07	SP = 08	SP = 09	SP = 0A

Popping from the stack

Popping the contents of the stack back into a given register is the opposite process of pushing. With every pop, the top byte of the stack is copied to the register specified by the instruction and the stack pointer is decremented once. Example 2-9 demonstrates the POP instruction.

The upper limit of the stack

As mentioned earlier, in the 8051 RAM locations 08 to 1F can be used for the stack. This is due to the fact that locations 20 - 2FH of RAM are reserved for bit-addressable memory and must not be used by the stack. If in a given program we need more than 24 bytes (08 to 1FH = 24 bytes) of stack, we can change the SP to point to RAM locations 30 - 7FH. This is done with the instruction "MOV SP,#xx".

Example 2-9

Examining the stack, show the contents of the registers and SP after execution of the following instructions. All values are in hex.

```
POP  3     ;POP stack into R3
POP  5     ;POP stack into R5
POP  2     ;POP stack into R2
```

Solution:

	After POP 3	After POP 5	After POP 2
0B 54	0B	0B	0B
0A F9	0A F9	0A	0A
09 76	09 76	09 76	09
08 6C	08 6C	08 6C	08 6C
Start SP = 0B	SP = 0A	SP = 09	SP = 08

CALL instruction and the stack

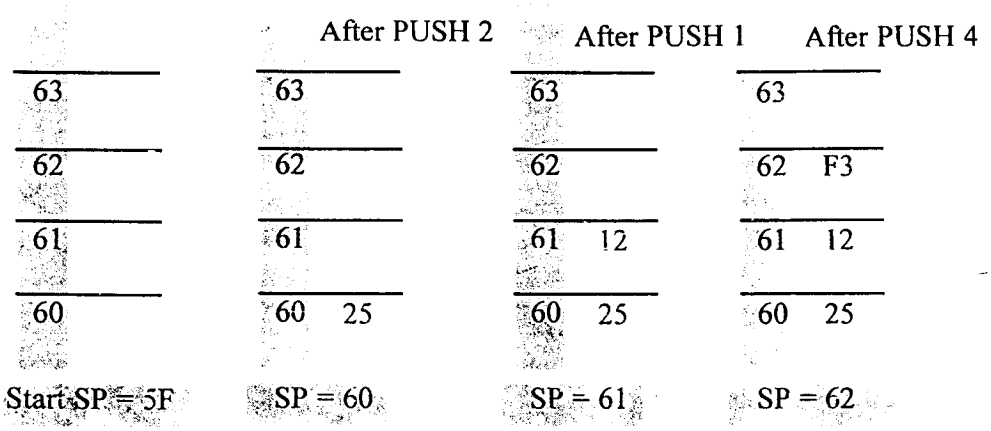
In addition to using the stack to save registers, the CPU also uses the stack to save the address of the instruction just below the CALL instruction. This is how the CPU knows where to resume when it returns from the called subroutine. More information on this will be given in Chapter 3 when we discuss the CALL instruction.

Example 2-10

Show the stack and stack pointer for the following instructions.

```
MOV  SP,#5FH    ;make RAM location 60H
                ;first stack location
MOV  R2,#25H
MOV  R1,#12H
MOV  R4,#0F3H
PUSH 2
PUSH 1
PUSH 4
```

Solution:



Stack and bank 1 conflict

Recall from our earlier discussion that the stack pointer register points to the current RAM location available for the stack. As data is pushed onto the stack, SP is incremented. Conversely, it is decremented as data is popped off the stack into the registers. The reason that the SP is incremented after the push is to make sure that the stack is growing toward RAM location 7FH, from lower addresses to upper addresses. If the stack pointer were decremented after push instructions, we would be using RAM locations 7, 6, 5, etc. which belong to R7 to R0 of bank 0, the default register bank. This incrementing of the stack pointer for push instructions also ensures that the stack will not reach location 0 at the bottom of RAM, and consequently run out of space for the stack. However, there is a problem with the default setting of the stack. Since SP = 07 when the 8051 is powered up, the first location of the stack is RAM location 08 which also belongs to register R0 of register bank 1. In other words, register bank 1 and the stack are using the same memory space. If in a given program we need to use register banks 1 and 2, we can reallocate another section of RAM to the stack. For example, we can allocate RAM locations 60H and higher to the stack as shown in Example 2-10.

Review Questions

1. What is the size of the SP register?
2. With each PUSH instruction, the stack pointer register, SP, is _____ (incremented, decremented) by 1.
3. With each POP instruction, the SP is _____ (incremented, decremented) by 1.
4. On power-up, the 8051 uses RAM location _____ as the first location of the stack.
5. On power up, the 8051 uses bank ___ for registers R0 - R7.
6. On power up, the 8051 uses RAM locations _____ to _____ for registers R0 - R7 (register bank 0).
7. Which register bank is used if we alter RS0 and RS1 of the PSW by the following two instructions?

```
SETB PSW.3
SETB PSW.4
```

8. In Question 7, what RAM locations are used for register R0 - R7?

SUMMARY

This chapter began with an exploration of the major registers of the 8051, including A, B, R0, R1, R2, R3, R4, R5, R6, R7, DPTR, and PC. The use of these registers was demonstrated in the context of programming examples. The process of creating an Assembly language program was described from writing the source file, to assembling it, linking, and executing the program. The PC (program counter) register always points to the next instruction to be executed. The way the 8051 uses program ROM space was explored because 8051 Assembly language programmers must be aware of where programs are placed in ROM, and how much memory is available.

An Assembly language program is composed of a series of statements that are either instructions or pseudo-instructions, also called *directives*. Instructions are translated by the assembler into machine code. Pseudo-instructions are not translated into machine code: They direct the assembler in how to translate instructions into machine code. Some pseudo-instructions, called *data directives*, are used to define data. Data is allocated in byte-size increments. The data can be in binary, hex, decimal, or ASCII formats.

Flags are useful to programmers since they indicate certain conditions, such as carry or overflow, that result from execution of instructions. The stack is used to store data temporarily during execution of a program. The stack resides in the RAM space of the 8051, which was diagrammed and explained. Manipulation of the stack via POP and PUSH instructions was also explored.

PROBLEMS

SECTION 2.1: INSIDE THE 8051

1. Most registers in the 8051 are _____ bits wide.
2. Registers R0 - R7 are all _____ bits wide
3. Registers ACC and B are _____ bits wide.
4. Name a 16-bit register in the 8051.
5. To load R4 with the value 65H, the pound sign is _____ (necessary, optional) in the instruction "`MOV R4,#65H`".
6. What is the result of the following code and where is it kept?

```
MOV  A,#15H
MOV  R2,#13H
ADD  A,R2
```

7. Which of the following is (are) illegal?
   (a) `MOV R3,#500` (b) `MOV R1,#50` (c) `MOV R7,#00`
   (d) `MOV A,#255H` (e) `MOV A,#50H` (f) `MOV A,#F5H`
   (g) `MOV R9,#50H`
8. Which of the following is (are) illegal?
   (a) `ADD R3,#50H` (b) `ADD A,#50H` (c) `ADD R7,R4`
   (d) `ADD A,#255H` (e) `ADD A,R5` (f) `ADD A,#F5H`
   (g) `ADD R3,A`
9. What is the result of the following code and where is it kept?

```
MOV  R4,#25H
MOV  A,#1FH
ADD  A,R4
```

10. What is the result of the following code and where is it kept?

```
MOV  A,#15
MOV  R5,#15
ADD  A,R5
```

SECTION 2.2: INTRODUCTION TO 8051 ASSEMBLY PROGRAMMING and

SECTION 2.3: ASSEMBLING AND RUNNING AN 8051 PROGRAM

11. Assembly language is a_____ (low, high) level language while C is a _____ (low, high) level language.
12. Of C and Assembly language, which is more efficient in terms of code generation (i.e., the amount of ROM space it uses)?
13. Which program produces the "obj" file?
14. True or false. The source file has the extension "src" or "asm".
15. Which file provides the listing of error messages?
16. True or false. The source code file can be a non-ASCII file.
17. True or false. Every source file must have ORG and END directives.
18. Do the ORG and END directives produce opcodes?
19. Why are the ORG and END directives also called pseudocode?
20. True or false. The ORG and END directives appear in the ".lst" file.

SECTION 2.4: THE PROGRAM COUNTER AND ROM SPACE IN THE 8051

21. Every 8051 family member wakes up at address _____ when it is powered up.
22. A programmer puts the first opcode at address 100H. What happens when the microcontroller is powered up?
23. Find the number of bytes each of the following instructions take.
    (a) `MOV A,#55H` (b) `MOV R3,#3` (c) `INC R2`
    (d) `ADD A,#0` (e) `MOV A,R1` (f) `MOV R3,A`
    (g) `ADD A,R2`
24. Pick up a program listing of your choice, and show the ROM memory addresses and their contents.
25. Find the address of the last location of on-chip ROM for each of the following.
    (a) DS5000-16 (b) DS5000-8 (c) DS5000-32
    (d) AT89C52 (e) 8751 (f) AT89C51
    (g) DS5000-64
26. Show the lowest and highest values (in hex) that the 8051 program counter can take.
27. A given 8051 has 7FFFH as the address of its last location of on-chip ROM. What is the size of on-chip ROM for this 8051?
28. Repeat Question 27 for 3FFH.

SECTION 2.5: 8051 DATA TYPES AND DIRECTIVES

29. Compile and state the contents of each ROM location for the following data.

```
            ORG   200H
MYDAT_1:    DB    "Earth"
MYDAT_2:    DB    "987-65"
MYDAT_3:    DB    "GABEH 98"
```

30. Compile and state the contents of each ROM location for the following data.

```
            ORG   340H
DAT_1:      DB    22,56H,10011001B,32,0F6H,11111011B
```

SECTION 2.6: 8051 FLAG BITS AND THE PSW REGISTER

31. The PSW is a(n) _____ -bit register.
32. Which bits of PSW are used for the CY and AC flag bits, respectively?
33. Which bits of PSW are used for the OV and P flag bits, respectively?
34. In the ADD instruction, when is CY raised?
35. In the ADD instruction, when is AC raised?
36. What is the value of the CY flag after the following code?

```
CLR  C          ;CY = 0
CPL  C          ;complement carry
```

37. Find the CY flag value after each of the following codes.

```
(a) MOV A,#54H      (b) MOV A,#00        (c) MOV A,#250
    ADD A,#0C4H         ADD A,#0FFH          ADD A,#05
```

38. Write a simple program in which the value 55H is added 5 times.

39. Which bits of the PSW are responsible for selection of the register banks?
40. On power up, what is the location of the first stack?
41. In the 8051, which register bank conflicts with the stack?
42. In the 8051, what is the size of the stack pointer (SP) register?
43. On power up, which of the register banks is used ?
44. Give the address locations of RAM assigned to various banks.
45. Assuming the use of bank 0, find at what RAM location each of the following lines stored the data.
    (a) `MOV R4,#32H` (b) `MOV R0,#12H`
    (c) `MOV R7,#3FH` (d) `MOV R5,#55H`
46. Repeat Problem 45 for bank 2.
47. After power up, show how to select bank 2 with a single instruction.
48. Show the stack and stack pointer for each line of the following program.

```
        ORG  0
MOV     R0,#66H
MOV     R3,#7FH
MOV     R7,#5DH
PUSH    0
PUSH    3
PUSH    7
CLR     A
MOV     R3,A
MOV     R7,A
POP     3
POP     7
POP     0
```

49. In Problem 48, does the sequence of POP instructions restore the original values of registers R0, R3, and R7? If not, show the correct sequence of instructions.
50. Show the stack and stack pointer for each line of the following program.

```
        ORG  0
MOV     SP,#70H
MOV     R5,#66H
MOV     R2,#7FH
MOV     R7,#5DH
PUSH    5
PUSH    2
PUSH    7
CLR     A
MOV     R2,A
MOV     R7,A
POP     7
POP     2
POP     5
```

ANSWERS TO REVIEW QUESTIONS

SECTION 2.1: INSIDE THE 8051

1. MOV A,#34H
   MOV B,#3FH
   ADD A,B
2. MOV A,#16H
   ADD A,#0CDH
   MOV R2,A
3. False
4. FF hex and 255 in decimal
5. 8

SECTION 2.2: INTRODUCTION TO 8051 ASSEMBLY PROGRAMMING

1. The real work is performed by instructions such as MOV and ADD. Pseudo-instructions, also called assembly directives, instruct the assembler in doing its job.
2. The instruction mnemonics, pseudo-instructions
3. False
4. All except (c)
5. Assembler directive
6. True
7. (c)

SECTION 2.3: ASSEMBLING AND RUNNING AN 8051 PROGRAM

1. True 2. True 3. (a) 4. (b) and (d) 5. (d)

SECTION 2.4: THE PROGRAM COUNTER AND ROM SPACE IN THE 8051

1. 16 2. True 3. 0000H 4. 2
5. With 8K bytes, we have 8192 (8 × 1024 = 8192) bytes, and the ROM space is 0000 to 1FFFH.

SECTION 2.5: 8051 DATA TYPES AND DIRECTIVES

1. DB 2. 7
3. If the value is to be changed later, it can be done once in one place instead of at every occurrence.
4. (a) 4 bytes (b) 7 bytes
5. This places the ASCII values for each character in memory locations starting at 200H. Notice that all values are in hex.
   200 = (41)
   201 = (42)
   202 = (43)
   203 = (31)
   204 = (32)
   205 = (33)

SECTION 2.6: 8051 FLAG BITS AND THE PSW REGISTER

1. PSW (program status register) 2. 8 bits
3. D1 and D5 which are referred to as PSW.1 and PSW.5, respectively.
4.

```
   Hex          binary
    FF          1111 1111
  +  1        +        1
   100         10000 0000   This leads to CY=1 and AC=1
```

5.

```
   Hex       binary
   C2        1100 0010
 + 3D      + 0011 1101
   FF        1111 1111
```

This leads to CY = 0 and AC = 0.

SECTION 2.7: 8051 REGISTER BANKS AND STACK

1. 8-bit 2. Incremented 3. Decremented
4. 08 5. 0 6. 0 - 7
7. Register bank 3
8. RAM locations 18H to 1FH

CHAPTER 3

JUMP, LOOP, AND CALL INSTRUCTIONS

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **Code 8051 Assembly language instructions using loops**
- **Code 8051 Assembly language conditional jump instructions**
- **Explain conditions that determine each conditional jump instruction**
- **Code long jump instructions for unconditional jumps**
- **Code short jump instructions for unconditional short jumps**
- **Calculate target addresses for jump instructions**
- **Code 8051 subroutines**
- **Describe precautions in using the stack in subroutines**
- **Discuss crystal frequency versus machine cycle**
- **Code 8051 programs to generate a time delay**

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 8051 to achieve this. This chapter covers the control transfer instructions available in 8051 Assembly language. In the first section we discuss instructions used for looping, as well as instructions for conditional and unconditional jumps. In the second section we examine CALL instructions and their uses. In the third section, time delay subroutines are described.

SECTION 3.1: LOOP AND JUMP INSTRUCTIONS

In this section we first discuss how to perform a looping action in the 8051 and then talk about jump instructions, both conditional and unconditional.

Looping in the 8051

Repeating a sequence of instructions a certain number of times is called a *loop*. The loop is one of most widely used actions that any microprocessor performs. In the 8051, the loop action is performed by the instruction "`DJNZ reg, label`". In this instruction, the register is decremented; if it is not zero, it jumps to the target address referred to by the label. Prior to the start of the loop the register is loaded with the counter for the number of repetitions. Notice that in this instruction both the register decrement and the decision to jump are combined into a single instruction.

Example 3-1

Write a program to
(a) clear ACC, then
(b) add 3 to the accumulator ten times.

Solution:

```
;This program adds value 3 to the ACC ten times

          MOV   A,#0       ;A=0, clear ACC
          MOV   R2,#10     ;load counter R2=10
AGAIN:    ADD   A,#03      ;add 03 to ACC
          DJNZ  R2,AGAIN   ;repeat until R2=0(10 times)
          MOV   R5,A       ;save A in R5
```

In the program in Example 3-1, the R2 register is used as a counter. The counter is first set to 10. In each iteration the instruction DJNZ decrements R2 and checks its value. If R2 is not zero, it jumps to the target address associated with label "AGAIN". This looping action continues until R2 becomes zero. After R2 becomes zero, it falls through the loop and executes the instruction immediately below it, in this case the "`MOV R5,A`" instruction.

Notice in the DJNZ instruction that the registers can be any of R0 - R7. The counter can also be a RAM location as we will see in Chapter 5.

Example 3-2

What is the maximum number of times that the loop in Example 3-1 can be repeated?

Solution:

Since R2 holds the count and R2 is an 8-bit register, it can hold a maximum of FFH (255 decimal); therefore, the loop can be repeated a maximum of 256 times.

Loop inside a loop

As shown in Example 3-2, the maximum count is 256. What happens if we want to repeat an action more times than 256? To do that, we use a loop inside a loop, which is called a *nested loop*. In a nested loop, we use two registers to hold the count. See Example 3-3.

Example 3-3

Write a program to (a) load the accumulator with the value 55H, and (b) complement the ACC 700 times.

Solution:

Since 700 is larger than 255 (the maximum capacity of any register), we use two registers to hold the count. The following code shows how to use R2 and R3 for the count.

```
        MOV  A,#55H     ;A=55H
        MOV  R3,#10     ;R3=10, the outer loop count
NEXT:   MOV  R2,#70     ;R2=70, the inner loop count
AGAIN:  CPL  A          ;complement A register
        DJNZ R2,AGAIN   ;repeat it 70 times (inner loop)
        DJNZ R3,NEXT
```

In this program, R2 is used to keep the inner loop count. In the instruction "DJNZ R2,ACAIN", whenever R2 becomes 0 it falls through and "DJNZ R3,NEXT" is executed. This instruction forces the CPU to load R2 with the count 70 and the inner loop starts again. This process will continue until R3 becomes zero and the outer loop is finished.

Other conditional jumps

Conditional jumps for the 8051 are summarized in Table 3-1. More details of each instruction are provided in Appendix A. In Table 3-1, notice that some of the instructions, such as JZ (jump if A = zero) and JC (jump if carry), jump only if a certain condition is met. Next we examine some conditional jump instructions with examples.

JZ (jump if A = 0)

In this instruction the content of register A is checked. If it is zero, it jumps to the target address. For example, look at the following code.

```
        MOV   A,R0          ;A=R0
        JZ    OVER          ;jump if A = 0
        MOV   A,R1          ;A=R1
        JZ .  OVER          ;jump if A = 0
        ...
OVER:
```

In this program, if either R0 or R1 is zero, it jumps to the label OVER. Notice that the JZ instruction can be used only for register A. It can only check to see whether the accumulator is zero, and it does not apply to any other register. More importantly, you don't have to perform an arithmetic instruction such as decrement to use the JNZ instruction. See Example 3-4.

Table 3-1: 8051 Conditional Jump Instructions

Instruction	Action
JZ	Jump if A = 0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and jump if A ≠ 0
CJNE A,byte	Jump if A ≠ byte
CJNE reg,#data	Jump if byte ≠ #data
JC	Jump if CY = 1
JNC	Jump if CY = 0
JB	Jump if bit = 1
JNB	Jump if bit = 0
JBC	Jump if bit = 1 and clear bit

Example 3-4

Write a program to determine if R5 contains the value 0. If so, put 55H in it.

Solution:

```
        MOV   A,R5        ;copy R5 to A
        JNZ   NEXT        ;jump if A is not zero
        MOV R5,#55H
NEXT:   ...
```

JNC (jump if no carry, jumps if CY = 0)

In this instruction, the carry flag bit in the flag (PSW) register is used to make the decision whether to jump. In executing "`JNC label`", the processor looks at the carry flag to see if it is raised (CY = 1). If it is not, the CPU starts to fetch and execute instructions from the address of the label. If CY = 1, it will not jump but will execute the next instruction below JNC.

It needs to be noted that there is also a "`JC label`" instruction. In the JC instruction, if CY = 1 it jumps to the target address. We will give more examples of these instructions in the context of applications in future chapters.

There is also a JB (jump if bit is high) and JNB (jump if bit is low). These are discussed in Chapters 4 and 8 when bit manipulation instructions are discussed.

Example 3-5

Find the sum of the values 79H, F5H, and E2H. Put the sum in registers R0 (low byte) and R5 (high byte).

Solution:

```
        MOV   A,#0        ;clear A(A=0)
        MOV   R5,A        ;clear R5
        ADD   A,#79H      ;A=0+79H=79H
        JNC   N_1         ;if no carry, add next number
        INC   R5          ;if CY=1, increment R5
N_1:    ADD   A,#0F5H     ;A=79+F5=6E and CY=1
        JNC   N_2         ;jump if CY=0
        INC   R5          ;If CY=1 then increment R5(R5=1)
N_2:    ADD   A,#0E2H     ;A=6E+E2=50 and CY=1
        JNC   OVER        ;jump if CY=0
        INC   R5          ;if CY=1, increment 5
OVER:MOV  R0,A            ;Now R0=50H, and R5=02
```

All conditional jumps are short jumps

It must be noted that all conditional jumps are short jumps, meaning that the address of the target must be within −128 to +127 bytes of the contents of the program counter (PC). This very important concept is discussed at the end of this section.

Unconditional jump instructions

The unconditional jump is a jump in which control is transferred unconditionally to the target location. In the 8051 there are two unconditional jumps: LJMP (long jump) and SJMP (short jump). Each is discussed below.

LJMP (long jump)

LJMP is an unconditional long jump. It is a 3-byte instruction in which the first byte is the opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address allows a jump to any memory location from 0000 to FFFFH.

Remember that although the program counter in the 8051 is 16-bit, thereby giving a ROM address space of 64K bytes, not all 8051 family members have that much on-chip program ROM. The original 8051 had only 4K bytes of on-chip ROM for program space; consequently, every byte was precious. For this reason there is also a SJMP (short jump) instruction which is a 2-byte instruction as opposed to the 3-byte LJMP instruction. This can save some bytes of memory in many applications where memory space is in short supply. SJMP is discussed next.

SJMP (short jump)

In this 2-byte instruction, the first byte is the opcode and the second byte is the relative address of the target location. The relative address range of 00 - FFH

is divided into forward and backward jumps; that is, within -128 to +127 bytes of memory relative to the address of the current PC (program counter). If the jump is forward, the target address can be within a space of 127 bytes from the current PC. If the target address is backward, the target address can be within -128 bytes from the current PC. This is explained in detail next.

Calculating the short jump address

In addition to the SJMP instruction, all conditional jumps such as JNC, JZ, and DJNZ are also short jumps due to the fact that they are all two-byte instructions. In these instructions the first byte is the opcode and the second byte is the relative address. The target address is relative to the value of the program counter. To calculate the target address, the second byte is added to the PC of the instruction immediately below the jump. To understand this, look at Example 3-6.

Example 3-6

Using the following list file, verify the jump forward address calculation.

```
Line  PC    Opcode          Mnemonic  Operand
01    0000                  ORG       0000
02    0000  7800            MOV       R0,#0
03    0002  7455            MOV       A,#55H
04    0004  6003            JZ        NEXT
05    0006  08              INC       R0
06    0007  04      AGAIN:  INC       A
07    0008  04              INC       A
08    0009  2477    NEXT:   ADD       A,#77h
09    000B  5005            JNC       OVER
10    000D  E4              CLR       A
11    000E  F8              MOV       R0,A
12    000F  F9              MOV       R1,A
13    0010  FA              MOV       R2,A
14    0011  FB              MOV       R3,A
15    0012  2B      OVER:   ADD       A,R3
16    0013  50F2            JNC       AGAIN
17    0015  80FE    HERE:   SJMP      HERE
18    0017                  END
```

Solution:

First notice that the JZ and JNC instructions both jump forward. The target address a forward jump is calculated by adding the PC of the following instruction to the s ond byte of the short jump instruction, which is called the relative address. In line 4 instruction "JZ NEXT" has opcode of 60 and operand of 03 at the addresses of 0004 a 0005. The 03 is the relative address, relative to the address of the next instruction I R0, which is 0006. By adding 0006 to 3, the target address of the label NEXT, whic 0009, is generated. In the same way for line 9, the "JNC OVER" instruction has opc and operand of 50 and 05 where 50 is the opcode and 05 the relative address. Therefo 05 is added to 000D, the address of instruction "CLR A", giving 12H, the address label OVER.

Example 3-7

Verify the calculation of backward jumps in Example 3-6.

Solution:

In that program list, "JNC AGAIN" has opcode 50 and relative address F2H. When the relative address of F2H is added to 15H, the address of the instruction below the jump, we have 15H + F2H = 07 (the carry is dropped). Notice that 07 is the address of label AGAIN. Look also at "SJMP HERE", which has 80 and FE for the opcode and relative address, respectively. The PC of the following instruction, 0017H, is added to FEH, the relative address, to get 0015H, address of the HERE label (17H + FEH = 15H). Notice that FEH is -2 and 17H + (-2) = 15H. For further discussion of the addition of negative numbers, see Chapter 6.

Jump backward target address calculation

While in the case of a forward jump, the displacement value is a positive number between 0 to 127 (00 to 7F in hex), for the backward jump the displacement is a negative value of 0 to -128 as explained in Example 3-7.

It must be emphasized that regardless of whether the SJMP is a forward or backward jump, for any short jump the address of the target address can never be more than -128 to +127 bytes from the address associated with the instruction below the SJMP. If any attempt is made to violate this rule, the assembler will generate an error stating the jump is out of range.

Review Questions

1. The mnemonic DJNZ stands for _____.
2. True or false. "DJNZ R5, BACK" combines a decrement and a jump in a single instruction.
3. "JNC HERE" is a 2 -byte instruction.
4. In "JZ NEXT", which register's content is checked to see if it is zero? (A)
5. LJMP is a 3 -byte instruction.

SECTION 3.2: CALL INSTRUCTIONS

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space. In the 8051 there are two instructions for call: LCALL (long call) and ACALL (absolute call). Deciding which one to use depends on the target address. Each instruction is explained next.

LCALL (long call)

In this 3-byte instruction, the first byte is the opcode and the second and third bytes are used for the address of the target subroutine. Therefore, LCALL can be used to call subroutines located anywhere within the 64K byte address space of

the 8051. To make sure that after execution of the called subroutine the 8051 knows where to come back to, it automatically saves on the stack the address of the instruction immediately below the LCALL. When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instructions from the new location. After finishing execution of the subroutine, the instruction RET (return) transfers control back to the caller. Every subroutine needs RET as the last instruction. See Example 3-8.

The following points should be noted for the program in Example 3-8.

1. Notice the DELAY subroutine. Upon executing the first "LCALL DELAY", the address of the instruction right below it, "MOV A, #0AAH", is pushed onto the stack, and the 8051 starts to execute instructions at address 300H.
2. In the DELAY subroutine, first the counter R5 is set to 255 (R5 = FFH); therefore, the loop is repeated 256 times. When R5 becomes 0, control falls to the RET instruction which pops the address from the stack into the program counter and resumes executing the instructions after the CALL.

Example 3-8

Write a program to toggle all the bits of port 1 by sending to it the values 55H and AAH continuously. Put a time delay in between each issuing of data to port 1. This program will be used to test the ports of the 8051 in the next chapter.

Solution:

```
        ORG     0
BACK:   MOV     A,#55H      ;load A with 55H
        MOV     P1,A        ;send 55H to port 1
        LCALL   DELAY       ;time delay
        MOV     A,#0AAH     ;load A with AA (in hex)
        MOV     P1,A        ;send AAH to port 1
        LCALL   DELAY
        SJMP    BACK        ;keep doing this indefinitely
;-------this is the delay subroutine
        ORG     300H        ;put time delay at address 300H
DELAY:  MOV     R5,#0FFH    ;R5=255(FF in hex),the counter
AGAIN:  DJNZ    R5,AGAIN    ;stay here until R5 becomes 0
        RET                 ;return to caller (when R5 = 0)
        END                 ;end of asm file
```

The amount of time delay in Example 3-8 depends on the frequency of the 8051. How to calculate the exact time will be explained in detail in Chapter 4. However you can increase the time delay by using a nested loop as shown below.

```
DELAY:                      ;nested loop delay
        MOV   R4,#255       ;R4=255(FF in hex)
NEXT:   MOV   R5,#255       ;R5=255(FF in hex)
AGAIN:  DJNZ  R5,AGAIN      ;stay here until R5 becomes 0
        DJNZ  R4,NEXT       ;decrement R4
                            ;keep loading R5 until R4=0
        RET                 ;return (when R4 = 0)
```

CALL instruction and the role of the stack

The stack and stack pointer were covered in the last chapter. To understand the importance of the stack in microcontrollers, we now examine the contents of the stack and stack pointer for Example 3-8. This is shown in Example 3-9.

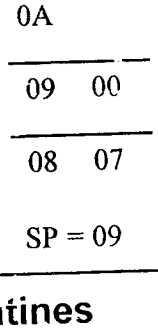
Example 3-9

Analyze the stack contents after the execution of the first LCALL in the following.

Solution:

```
001   0000                      ORG   0
002   0000 7455   BACK:         MOV   A,#55H   ;load A with 55H
003   0002 F590                 MOV   P1,A     ;send 55H to port 1
004   0004 120300               LCALL DELAY    ;time delay
005   0007 74AA                 MOV    A,#0AAH;load A with AAH
006   0009 F590                 MOV   P1,A     ;send AAH to port 1
007   000B 120300               LCALL DELAY
008   000E 80F0                 SJMP  BACK      ;keep doing this
009   0010
010   0010 ;----------this is the delay subroutine
011   0300                      ORG 300H
012   0300          DELAY:
013   0300 7DFF                 MOV   R5,#0FFH  ;R5=255
014   0302 DDFE   AGAIN:        DJNZ  R5,AGAIN  ;stay here
015   0304 22                   RET             ;return to caller
016   0305                      END             ;end of asm file
```

When the first LCALL is executed, the address of the instruction "MOV A,#0AAH" is saved on the stack. Notice that the low byte goes first and the high byte is last. The last instruction of the called subroutine must be a RET instruction which directs the CPU to POP the top bytes of the stack into the PC and resume executing at address 07. The diagram shows the stack frame after the first LCALL.



Use of PUSH and POP instructions in subroutines

Upon calling a subroutine, the stack keeps track of where the CPU should return after completing the subroutine. For this reason, we must be very careful in any manipulation of stack contents. The rule is that the number of PUSH and POP instructions must always match in any called subroutine. In other words, for every PUSH there must be a POP. See Example 3-10.

Calling subroutines

In Assembly language programming it is common to have one main program and many subroutines that are called from the main program. This allows you to make each subroutine into a separate module. Each module can be tested separately and then brought together with the main program. More importantly, in a large program the modules can be assigned to different programmers in order to shorten development time.

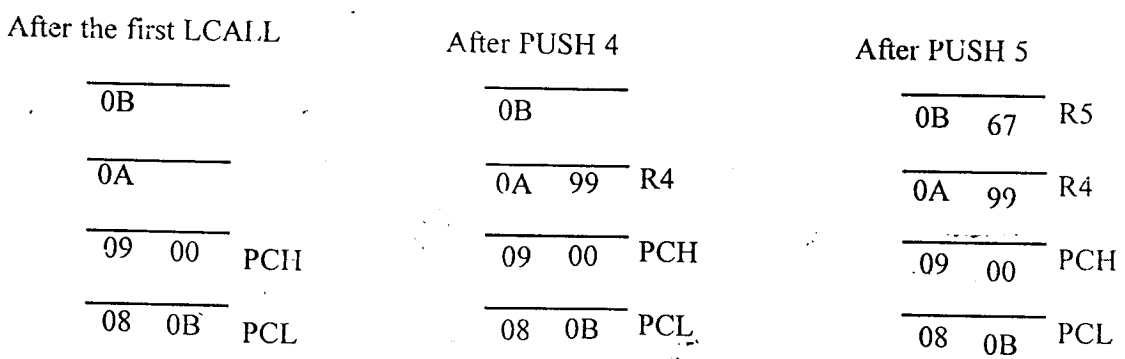
Example 3-10

Analyze the stack for the first LCALL instruction in the following program.

```
01 0000                ORG   0
02 0000 7455 BACK:     MOV   A,#55H      ;load A with 55H
03 0002 F590           MOV   P1,A        ;send 55H to port 1
04 0004 7C99           MOV   R4,#99H
05 0006 7D67           MOV   R5,#67H
06 0008 120300         LCALL DELAY       ;time delay
07 000B 74AA           MOV   A,#0AAH     ;Load A with AA
08 000D F590           MOV   P1,A        ;send AAH to port 1
09 000F 120300         LCALL DELAY
10 0012 80EC           SJMP  BACK        ;keep doing this
11 0014               ;--------this is the delay subroutine
12 0300                ORG 300H
13 0300 C004   DELAY:PUSH 4                ;PUSH R4
14 0302 C005           PUSH 5              ;PUSH R5
15 0304 7CFF           MOV  R4,#0FFH       ;R4=FFH
16 0306 7DFF   NEXT:   MOV  R5,#0FFH       ;R5=255
17 0308 DDFE   AGAIN:DJNZ R5,AGAIN
18 030A DCFA           DJNZ R4,NEXT
19 030C D005           POP  5              ;POP INTO R5
20 030E D004           POP  4              ;POP INTO R4
21 0310 22             RET                 ;return to caller
22 0311                END                 ;end of asm file
```

Solution:

First notice that for the PUSH and POP instructions we must specify the direct address of the register being pushed or popped. Here is the stack frame.



It needs to be emphasized that in using LCALL, the target address of the subroutine can be anywhere within the 64K bytes memory space of the 8051. This is not the case for the other call instruction, ACALL, which is explained next.

```
;MAIN program calling subroutines
          ORG  0
MAIN:     LCALL     SUBR_1
          LCALL     SUBR_2
          LCALL     SUBR_3

HERE:     SJMP      HERE
;---------end of MAIN
;
SUBR_1:   ....
          ....
          RET
;---------end of subroutine 1
;
SUBR_2:   ....
          ....
          RET
;---------end of subroutine 2

SUBR_3:   ....
          ....
          RET
;---------end of subroutine 3
          END       ;end of the asm file
```

Figure 3-1. 8051 Assembly Main Program That Calls Subroutines

ACALL (absolute call)

ACALL is a 2-byte instruction in contrast to LCALL, which is 3 bytes. Since ACALL is a 2-byte instruction, the target address of the subroutine must be within 2K bytes address because only 11 bits of the 2 bytes are used for the address. There is no difference between ACALL and LCALL in terms of saving the program counter on the stack or the function of the RET instruction. The only difference is that the target address for LCALL can be anywhere within the 64K byte address space of the 8051 while the target address of ACALL must be within a 2K-byte range. In many variations of the 8051 marketed by different companies, on-chip ROM is as low as 1K bytes. In such cases, the use of ACALL instead of LCALL can save a number of bytes of program ROM space.

Example 3-11

A developer is using the Atmel AT89C1051 microcontroller chip for a product. This chip has only 1K bytes of on-chip flash ROM. Which of the instructions LCALL and ACALL is most useful in programming this chip?

Solution:
The ACALL instruction is more useful since it is a 2-byte instruction. It saves one byte each time the call instruction is used.

Of course in addition to using compact instructions, we can program effeciently by having a detailed knowledge of all the instructions supported by a given microprocessor, and using them wisely. Look at Example 3-12.

Example 3-12

Rewrite Example 3-8 as efficiently as you can.

Solution:

```
          ORG   0
          MOV   A,#55H      ;load A with 55H
BACK:     MOV   P1,A        ;issue value in reg A to port 1
          ACALL DELAY       ;time delay
          CPL   A           ;complement reg A
          SJMP  BACK        ;keep doing this indefinitely

;----------this is the delay subroutine
DELAY:
          MOV   R5,#0FFH    ;R5=255(FF in hex),the counter
AGAIN:    DJNZ  R5,AGAIN    ;stay here until R5 becomes 0
          RET               ;return to caller
          END               ;end of asm file
```

Notice in this program that register A is set to 55H. By complementing 55H, we hav AAH; and by complementing AAH we have 55H. Why? "01010101" in binary (55H becomes "10101010" in binary (AAH) when it is complemented; and "10101010 becomes "01010101" if it is complemented.

Review Questions

1. What do the mnemonics "LCALL" and "ACALL" stand for?
2. True or false. In the 8051, control can be transferred anywhere within the 64K bytes of code space if using the LCALL instruction.
3. How does the CPU know where to return to after executing the RET instruction?
4. Describe briefly the function of the RET instruction.
5. The LCALL instruction is a ___-byte instruction.

SECTION 3.3: TIME DELAY GENERATION AND CALCULATION

In the last section we used the DELAY subroutine. How to generate various time delays and calculate exact delays is discussed in this section.

Machine cycle

For the CPU to execute an instruction takes a certain number of clock cycles. In the 8051 family, these clock cycles are referred to as *machine cycles*. Appendix A.2 provides the list of 8051 instructions and their machine cycles. To calculate a time delay, we use this list. In the 8051 family, the length of the machine cycle depends on the frequency of the crystal oscillator connected to the

8051 system. The crystal oscillator, along with on-chip circuitry, provide the clock source for the 8051 CPU (see Chapter 4). The frequency of the crystal connected to the 8051 family can vary from 4 MHz to 30 MHz, depending on the chip rating and manufacturer. Very often the 11.0592 MHz crystal oscillator is used to make the 8051-based system compatible with the serial port of the IBM PC (see Chapter 10). In the 8051, one machine cycle lasts 12 oscillator periods. Therefore, to calculate the machine cycle, we take 1/12 of the crystal frequency, then take its inverse, as shown in Example 3-13.

Example 3-13

The following shows crystal frequency for three different 8051-based systems. Find the period of the machine cycle in each case.
(a) 11.0592 MHz (b) 16 MHz (c) 20 MHz

Solution:

(a)11.0592/12 = 921.6 kHz; machine cycle is 1/921.6 kHz = 1.085 μs (microsecond)
(b) 16 MHz/12 = 1.333 MHz; machine cycle (MC) = 1/.1.333 MHz = 0.75 μs
(c) 20 MHz/12 = 1.66 MHz; MC = 1/1.66 MHz = 0.60 μs

Example 3-14

For an 8051 system of 11.0592 MHz, find how long it takes to execute each of the following instructions.

(a) `MOV R3,#55` (b) `DEC R3` (c) `DJNZ R2,target`
(d) `LJMP` (e) `SJMP` (f) `NOP` (no operation)
(g) `MUL AB`

Solution:

The machine cycle for a system of 11.0592 MHz is 1.085 μs as shown in Example 3-13. Table A-1 in Appendix A shows machine cycles for each of the above instructions. Therefore, we have:

Instruction	*Machine cycles*	*Time to execute*
(a) `MOV R3,#55`	1	1×1.085 μs = 1.085 μs
(b) `DEC R3`	1	1×1.085 μs = 1.085 μs
(c) `DJNZ R2,target`	2	2×1.085 μs = 2.17 μs
(d) `LJMP`	2	2×1.085 μs = 2.17 μs
(e) `SJMP`	2	2×1.085 μs = 2.17 μs
(f) `NOP`	1	1×1.085 μs = 1.085 μs
(g) `MUL AB`	4	4×1.085 μs = 4.34 μs

Delay calculation

As seen in the last section, a delay subroutine consists of two parts: (1) setting a counter, and (2) a loop. Most of the time delay is performed by the body of the loop, as shown in Example 3-15.

Example 3-15

Find the size of the delay in the following program, if the crystal frequency is 11.0592 MHz.

```
            MOV  A,#55H
AGAIN:      MOV  P1,A
            ACALL DELAY
            CPL  A
            SJMP AGAIN
;----Time delay
DELAY:      MOV  R3,#200
HERE:       DJNZ R3,HERE
            RET
```

Solution:

From Table A-1 in Appendix A, we have the following machine cycles for each instruction of the DELAY subroutine.

```
                                Machine Cycle
DELAY:      MOV  R3,#200            1
HERE:       DJNZ R3,HERE            2
            RET                     1
```

Therefore, we have a time delay of $[(200 \times 2) + 1 + 1] \times 1.085\ \mu s = 436.17\ \mu s$.

Very often we calculate the time delay based on the instructions inside the loop and ignore the clock cycles associated with the instructions outside the loop.

In Example 3-15, the largest value the R3 register can take is 255; therefore, one way to increase the delay is to use NOP instructions in the loop. NOP, which stands for "no operation," simply wastes time. This is shown in Example 3-16.

Loop inside loop delay

Another way to get a large delay is to use a loop inside a loop, which is also called a *nested loop*. See Example 3-17.

Example 3-16

Find the time delay for the following subroutine, assuming a crystal frequency of 11.0592 MHz.

```
                                  Machine Cycle
DELAY:    MOV  R3,#250              1

HERE:     NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          DJNZ R3,HERE              2

          RET                       1
```

Solution:

The time delay inside the HERE loop is [250 (1+1+1+1+2)] × 1.085 μs = 1500 × 1.085 μs = 1627.5 μs. Adding the two instructions outside the loop we have 1627.5 μs + 2 × 1.085 μ s = 1629.67 μs.

Example 3-17

For a machine cycle of 1.085 μs, find the time delay in the following subroutine.

```
DELAY:                            Machine Cycle
          MOV  R2,#200              1
AGAIN:    MOV  R3,#250              1
HERE:     NOP                       1
          NOP                       1
          DJNZ R3,HERE              2
          DJNZ R2,AGAIN             2
          RET                       1
```

For the HERE loop, we have (4 × 250) 1.085 μs = 1085 μs. The AGAIN loop repeats the HERE loop 200 times; therefore, we have 200 × 1085 μs = 217000, if we do not include the overhead. However, the instructions "MOV R3,#250" and "DJNZ R2,AGAIN" at the beginning and end of the AGAIN loop add (3 × 200 × 1.085 μs) = 651 μs to the time delay. As a result we have 217000 + 651 = 217651 μs = 217.651 milliseconds for total time delay associated with the above DELAY subroutine. Notice that in the case of a nested loop, as in all other time delay loops, the time is approximate since we have ignored the first and last instructions in the subroutine.

Review Questions

1. True or false. In the 8051, the machine cycle lasts 12 clock cycles of the crystal frequency.
2. The minimum number of machine cycles needed to execute an 8051 instruction is _____.
3. For Question 2, what is the maximum number of cycles needed, and for which instructions?
4. Find the machine cycle for a crystal frequency of 12 MHz.
5. Assuming a crystal frequency of 12 MHz, find the time delay associated with the loop section of the following DELAY subroutine.

```
DELAY:
            MOV   R3,#100
HERE:       NOP
            NOP
            NOP
            DJNZ  R3,HERE
            RET
```

SUMMARY

The flow of a program proceeds sequentially, from instruction to instruction, unless a control transfer instruction is executed. The various types of control transfer instructions in Assembly language include conditional and unconditional jumps, and call instructions.

The looping action in 8051 Assembly language is performed using a special instruction which decrements a counter and jumps to the top of the loop if the counter is not zero. Other jump instructions jump conditionally, based on the value of the carry flag, the accumulator, or bits of the I/O port. Unconditional jumps can be long or short, depending on the relative value of the target address. Special attention must be given to the effect of LCALL and ACALL instructions on the stack.

PROBLEMS

SECTION 3.1: LOOP AND JUMP INSTRUCTIONS

1. In the 8051, looping action with instruction "`DJNZ Rx,rel address`" is limited to ____ iterations.
2. If a conditional jump is not taken, what is the next instruction to be executed?
3. In calculating the target address for a jump, a displacement is added to the contents of register _____.
4. The mnemonic SJMP stands for _____ and it is a ___-byte instruction.
5. The mnemonic LJMP stands for _____ and it is a ___-byte instruction.
6. What is the advantage of using SJMP over LJMP?
7. True or false. The target of a short jump is within -128 to +127 bytes of the current PC.
8. True or false. All 8051 jumps are short jumps.

9. Which of the following instructions is (are) not a short jump?
   (a) JZ (b) JNC (c) LJMP (d) DJNZ
10. A short jump is a ___-byte instruction. Why?
11. True or false. All conditional jumps are short jumps.
12. Show code for a nested loop to perform an action 1000 times.
13. Show code for a nested loop to perform an action 100,000 times.
14. Find the number of times the following loop is performed.

```
        MOV  R6,#200
BACK:   MOV  R5,#100
HERE:   DJNZ R5,HERE
        DJNZ R6,BACK
```

15. The target address of a jump backward is a maximum of _____ bytes from the current PC.
16. The target address of a jump forward is a maximum of _____ bytes from the current PC.

SECTION 3.2: CALL INSTRUCTIONS

17. LCALL is a ___-byte instruction.
18. ACALL is a ___-byte instruction.
19. The ACALL target address is limited to ____ bytes from the present PC.
20. The LCALL target address is limited to ____ bytes from the present PC.
21. When LCALL is executed, how many bytes of the stack are used?
22. When ACALL is executed, how many bytes of the stack are used?
23. Why do the number of PUSH and POP instructions in a subroutine need to be equal?
24. Describe the action associated with the POP instruction.
25. Show the stack for the following code.

```
000B 120300          LCALL DELAY
000E 80F0            SJMP BACK     ;keep doing this
0010
0010 ;---------this is the delay subroutine
0300                 ORG 300H
0300       DELAY:
0300 7DFF            MOV  R5,#0FFH  ;R5=255
0302 DDFE  AGAIN:    DJNZ R5,AGAIN  ;stay here
0304 22              RET            ;return
```

26. Reassemble Example 3-10 at ORG 200 (instead of ORG 0) and show the stack frame for the first LCALL instruction.

SECTION 3.3:TIME DELAY GENERATION AND CALCULATION

27. Find the system frequency if the machine cycle = 1.2 μs.
28. Find the machine cycle if crystal frequency is 18 MHz.
29. Find the machine cycle if crystal frequency is 12 MHz.
30. Find the machine cycle if crystal frequency is 25 MHz.

31. True or false. LJMP and SJMP instructions take the same amount of time to execute even though one is a 3-byte instruction and the other one is a 2-byte instruction.

32. Find the time delay for the delay subroutine shown to the right, if the system frequency is 11.0592 MHz.

```
DELAY:    MOV  R3,#150
HERE:     NOP
          NOP
          NOP
          DJNZ R3,HERE
          RET
```

33. Find the time delay for the delay subroutine shown to the right, if the system frequency is 16 MHz.

```
DELAY:    MOV  R3,#200
HERE:     NOP
          NOP
          NOP
          DJNZ R3,HERE
          RET
```

34. Find the time delay for the delay subroutine shown to the right, if the system frequency is 11.0592 MHz.

```
DELAY:    MOV  R5,#100
BACK:     MOV  R2,#200
AGAIN:    MOV  R3,#250
HERE:     NOP
          NOP
          DJNZ R3,HERE
          DJNZ R2,AGAIN
          DJNZ R5,BACK
          RET
```

35. Find the time delay for the delay subroutine shown to the right, if the system frequency is 16 MHz.

```
DELAY:    MOV  R2,#150
AGAIN:    MOV  R3,#250
HERE:     NOP
          NOP
          NOP
          DJNZ R3,HERE
          DJNZ R2,AGAIN
          RET
```

ANSWERS TO REVIEW QUESTIONS

SECTION 3.1: LOOP AND JUMP INSTRUCTIONS

1. Decrement and jump if not zero 2. True 3. 2 4. A 5. 3

SECTION 3.2: CALL INSTRUCTIONS

1. Long CALL and Absolute CALL 2. True
3. The address of where to return is in the stack.
4. Upon executing the RET instruction, the CPU pops off the top two bytes of the stack into the program counter (PC) register and starts to execute from this new location.
5. 3

SECTION 3.3: TIME DELAY GENERATION AND CALCULATION

1. True 2. 1 3. MUL and DIV each take 4 machine cycles.
4. 12 MHz / 12 = 1 MHz, and MC = 1/1 MHz = 1 µs.
5. [100 (1+1+1+2)] × 1 µs = 500 µs = 0.5 milliseconds.

CHAPTER 4

I/O PORT PROGRAMMING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **Explain the purpose of each pin of the 8051 microcontroller**
- **List the 4 ports of the 8051**
- **Describe the dual role of port 0 in providing both data and addresses**
- **Code Assembly language to use the ports for input or output**
- **Explain the use of port 3 for interrupt signals**
- **Code 8051 instructions for I/O handling**
- **Code bit-manipulation instructions in the 8051**

This chapter describes the 8051 pins and then shows I/O port programming of the 8051 with many examples.

SECTION 4.1: PIN DESCRIPTION OF THE 8051

Although 8051 family members (e.g., 8751, 89C51, DS5000) come in different packages, such as DIP (dual in-line package), QFP (quad flat package), and LLC (leadless chip carrier), they all have 40 pins that are dedicated for various functions such as I/O, $\overline{RD}$, $\overline{WR}$, address, data, and interrupts. It must be noted that some companies provide a 20-pin version of the 8051 with a reduced number of I/O ports for less demanding applications. However, since the vast majority of developers use the 40-pin DIP package chip, we will concentrate on that.

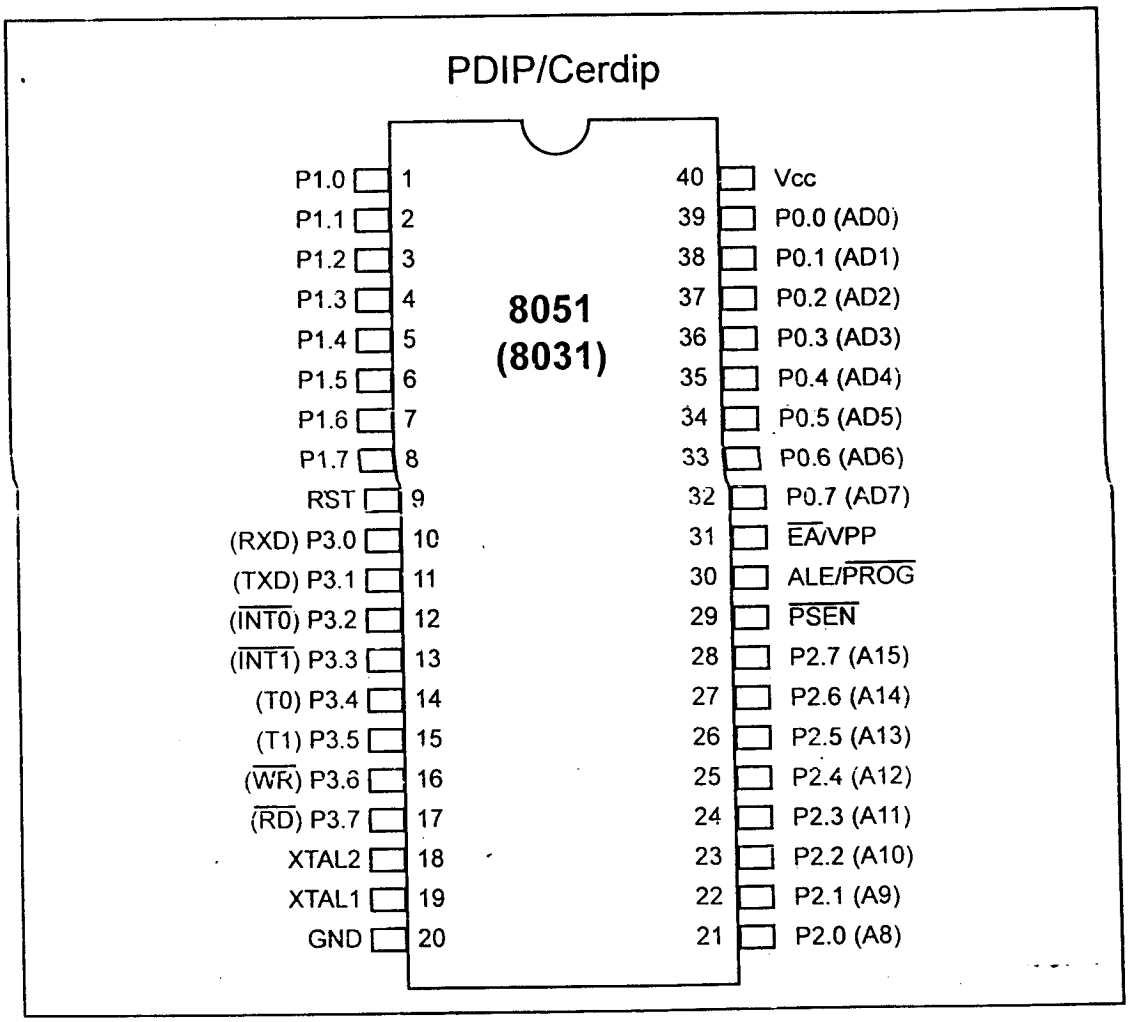


Figure 4-1. 8051 Pin Diagram

Examining Figure 4-1, note that of the 40 pins, a total of 32 pins are set aside for the four ports P0, P1, P2, and P3, where each port takes 8 pins. The rest of the pins are designated as V_{CC}, GND, XTAL1, XTAL2, RST, $\overline{EA}$, $\overline{PSEN}$. Of these 8 pins, six of them (V_{CC}, GND, XTAL1, XTAL2, RST, and $\overline{EA}$) are used by all members of the 8051 and 8031 families. In other words, they must be connected in order for the system to work, regardless of whether the microcontroller is of

the 8051 or 8031 family. The other two pins, $\overline{PSEN}$ and ALE, are used mainly in 8031-based systems. We first describe the function of each pin. Ports are discussed separately.

V_{CC}

Pin 40 provides supply voltage to the chip. The voltage source is +5V.

GND

Pin 20 is the ground.

XTAL1 and XTAL2

The 8051 has an on-chip oscillator but requires an external clock to run it. Most often a quartz crystal oscillator is connected to inputs XTAL1 (pin 19) and XTAL2 (pin 18). The quartz crystal oscillator connected to XTAL1 and XTAL2 also needs two capacitors of 30 pF value. One side of each capacitor is connected to the ground as shown in Figure 4-2 (a).

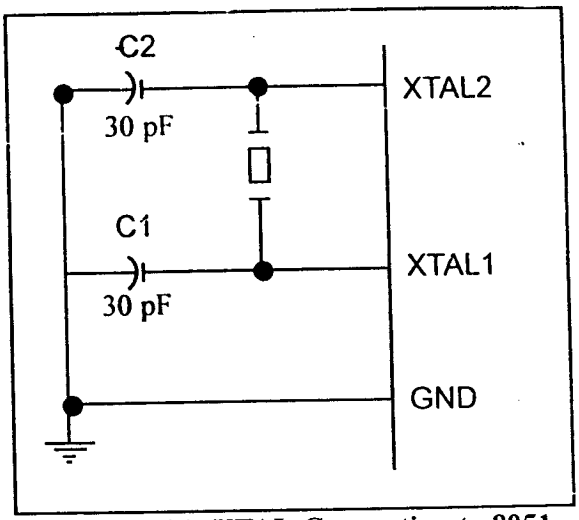


Figure 4-2 (a). XTAL Connection to 8051

It must be noted that there are various speeds of the 8051 family. Speed refers to the maximum oscillator frequency connected to XTAL. For example, a 12-MHz chip must be connected to a crystal with 12 MHz frequency or less. Likewise, a 20-MHz microcontroller requires a crystal frequency of no more than 20 MHz. When the 8051 is connected to a crystal oscillator and is powered up, we can observe the frequency on the XTAL2 pin using the oscilloscope.

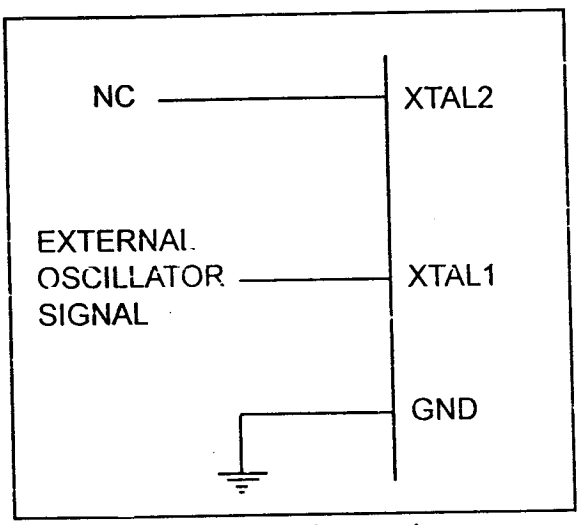


Figure 4-2 (b). XTAL Connection to an External Clock Source

If you decide to use a frequency source other than a crystal oscillator, such as a TTL oscillator, it will be connected to XTAL1; XTAL2 is left unconnected, as shown in Figure 4-2 (b).

Table 4-1: RESET Value of Some 8051 Registers

Register	Reset Value
PC	0000
ACC	0000
B	0000
PSW	0000
SP	0007
DPTR	0000

RST

Pin 9 is the RESET pin. It is an input and is active high (normally low). Upon applying a high puise to this pin, the microcontroller will reset and terminate all activities. This is often referred to

as a *power-on reset*. Activating a power-on reset will cause all values in the registers to be lost. Table 4-1 provides a partial list of 8051 registers and their values after power-on reset.

Notice that the value of the PC (program counter) is 0 upon reset, forcing the CPU to fetch the first opcode from ROM memory location 0000. This means that we must place the first line of source code in ROM location 0 because that is where the CPU wakes up and expects to find the first instruction. Figure 4-3 shows two ways of connecting the RST pin to the power-on reset circuitry.

In order for the RESET input to be effective, it must have a minimum duration of 2 machine cycles. In other words, the high pulse must be high for a minimum of 2 machine cycles before it is allowed to go low.

In the 8051, a machine cycle is defined as 12 oscillator periods, as discussed in Chapter 3, as shown again in Example 4-1.

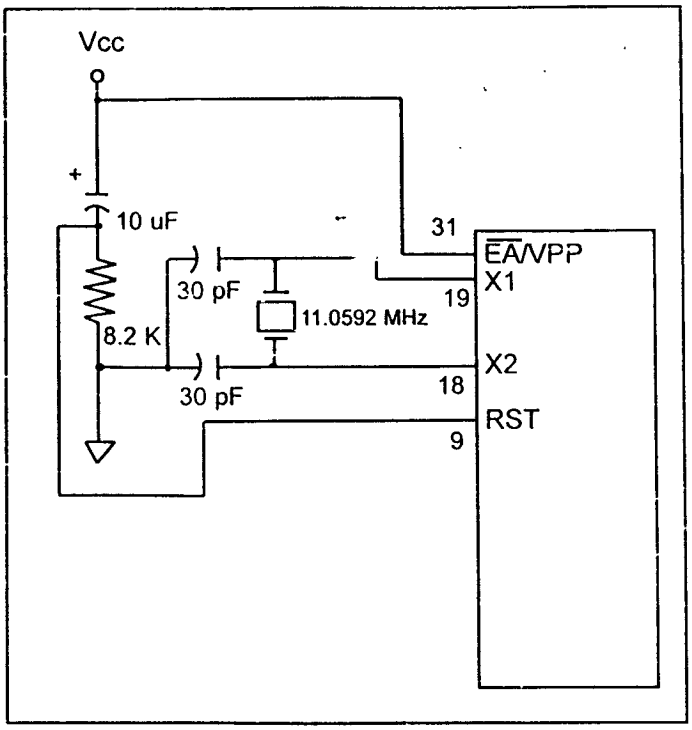


Figure 4-3 (a). Power-On RESET Circuit

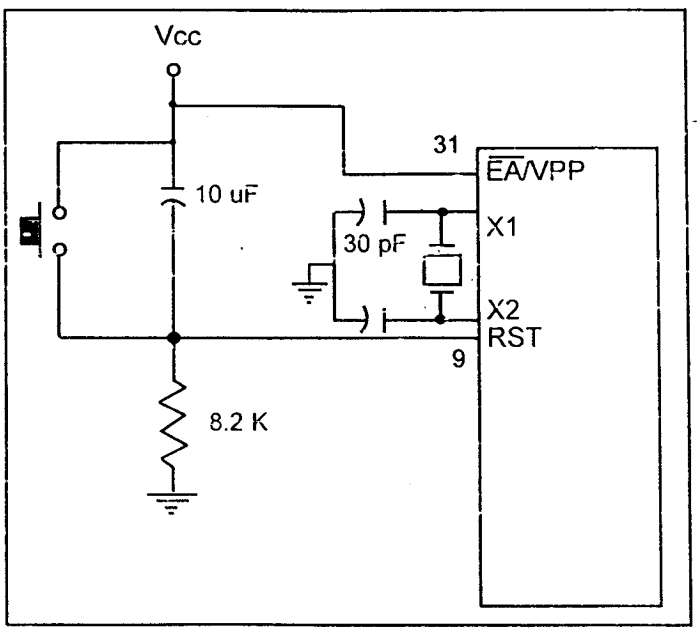


Figure 4-3 (b). Power-On RESET with Debounce

$\overline{EA}$

The 8051 family members, such as the 8751, 89C51, or DS5000, all come with on-chip ROM to store programs. In such cases, the $\overline{EA}$ pin is connected to V_{CC}. For family members such as the 8031 and 8032 in which there is no on-chip ROM, code is stored on an external ROM and is fetched by the 8031/32. Therefore, for the 8031 the $\overline{EA}$ pin must be connected to GND to indicate that the code is stored externally. $\overline{EA}$, which stands for "external access," is pin number 31 in the DIP packages. It is an input pin and must be connected to either V_{CC} or GND. In other words, it cannot be left unconnected.

In Chapter 14, we will show how the 8031 uses this pin along with $\overline{PSEN}$ to access programs stored in ROM memory located outside the 8031. In 8051 chips with on-chip ROM, such as the 8751, 89C51, or DS5000, $\overline{EA}$ is connected to V_{CC}, as we will see in the next section.

Example 4-1

Find the machine cycle for (a) XTAL = 11.0592 MHz (b) XTAL = 16 MHz.

Solution:
(a)11.0592 MHz / 12 = 921.6 kHz;
machine cycle = 1 / 921.6 kHz = 1.085 μs
(b)16 MHz / 12 = 1.333 MHz;
machine cycle = 1 / 1.333 MHz = 0.75 μs

The pins discussed so far must be connected no matter which family member is used. The next two pins are used mainly in 8031-based systems and are discussed in more detail in Chapter 14. The following is a brief description of each.

$\overline{PSEN}$

This is an output pin. $\overline{PSEN}$ stands for "program store enable." In an 8031-based system in which an external ROM holds the program code, this pin is connected to the OE pin of the ROM. See Chapter 14 to see how this is used.

ALE

ALE (address latch enable) is an output pin and is active high. When connecting an 8031 to external memory, port 0 provides both address and data. In other words, the 8031 multiplexes address and data through port 0 to save pins. The ALE pin is used for demultiplexing the address and data by connecting to the G pin of the 74LS373 chip. This is discussed in detail in Chapter 14.

I/O port pins and their functions

The four ports P0, P1, P2, and P3 each use 8 pins, making them 8-bit ports. All the ports upon RESET are configured as output, ready to be used as output ports. To use any of these ports as an input port, it must be programmed, as we will explain throughout this section. First, we describe each port.

Port 0

Port 0 occupies a total of 8 pins (pins 32 - 39). It can be used for input or output. To use the pins of port 0 as both input and output ports, each pin must be connected externally to a 10K ohm pull-up resistor. This is due to the fact that P0 is an open drain, unlike P1, P2, and P3, as we will soon see. *Open drain* is a term used for MOS chips in the same way that *open collector* is used for TTL chips. In any system using the 8751, 89C51, or DS5000 chips, we normally connect P0 to pull-up resistors. See Figure 4-4. In this way we take advantage of port 0 for both input and output. With external pull-up resistors connected upon reset, port 0 is configured as an output port. For example, the following code will continuously send out to port 0 the alternating values 55H and AAH.

```
        MOV     A,#55H
BACK:   MOV     P0,A
        ACALL   DELAY
        CPL     A
        SJMP    BACK
```

Port 0 as input

With resistors connected to port 0, in order to make it an input, the port must be programmed by writing 1 to all the bits. In the following code, port 0 is configured first as an input port by writing 1s to it, and then data is received from that port and sent to P1.

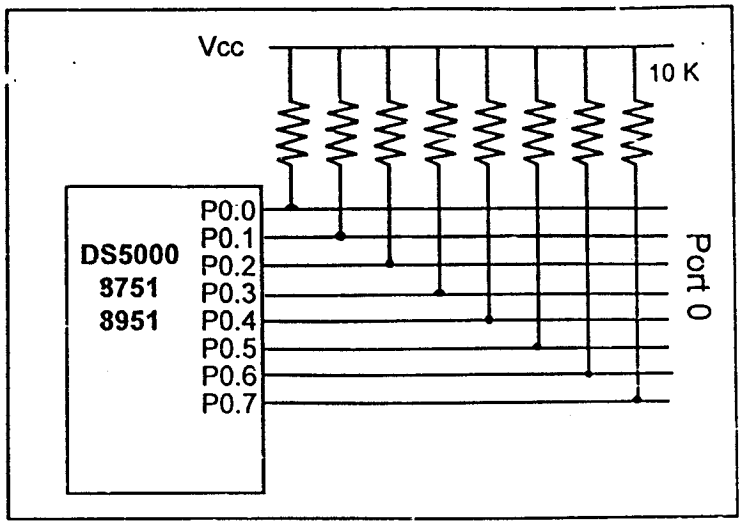


Figure 4-4. Port 0 with Pull-Up Resistors

```
        MOV  A,#0FFH    ;A = FF hex
        MOV  P0,A       ;make P0 an input port
                        ;by writing all 1s to it
BACK:   MOV  A,P0       ;get data from P0
        MOV  P1,A       ;send it to port 1
        SJMP BACK       ;keep doing it
```

Dual role of port 0

As shown in Figure 4-1, port 0 is also designated as AD0 - AD7, allowing it to be used for both address and data. When connecting an 8051/31 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save pins. ALE indicates if P0 has address or data. When ALE = 0, it provides data D0 - D7, but when ALE = 1 it has address A0 - A7. Therefore, ALE is used for demuliplexing address and data with the help of a 74LS373 latch, as we will see in Chapter 14.

Port 1

Port 1 occupies a total of 8 pins (pins 1 through 8). It can be used as input or output. In contrast to port 0, this port does not need any pull-up resistors since it already has pull-up resistors internally. Upon reset, port 1 is configured as an output port. For example, the following code will continuously send out to port 1 the alternating values 55H and AAH.

```
        MOV  A,#55H
BACK:   MOV  P1,A
        ACALL DELAY
        CPL  A
        SJMP BACK
```

Port 1 as input

To make port 1 an input port, it must programmed as such by writing 1 to all its bits. The reason for this is discussed in Appendix C.2. In the following code, port 1 is configured first as an input port by writing 1s to it, then data is received from that port and saved in R7, R6, and R5.

```
MOV   A,#0FFH    ;A=FF hex
MOV   P1,A       ;make P1 an input port
                 ;by writing all 1s to it
MOV   A,P1       ;get data from P1
MOV   R7,A       ;save it in reg R7
ACALL DEALY      ;wait
MOV   A,P1       ;get another data from P1
MOV   R6,A       ;save it in reg R6
ACALL DELAY      :wait
MOV   A,P1       ;get another data from P1
MOV   R5,A       ;save it in reg R5
```

Port 2

Port 2 occupies a total of 8 pins (pins 21 through 28). It can be used as input or output. Just like P1, port 2 does not need any pull-up resistors since it already has pull-up resistors internally. Upon reset, port 2 is configured as an output port. For example, the following code will send out continuously to port 2 the alternating values 55H and AAH. That is, all the bits of P2 toggle continuously.

```
        MOV   A,#55H
BACK:   MOV   P2,A
        ACALL DELAY
        CPL   A
        SJMP  BACK
```

Port 2 as input

To make port 2 an input, it must programmed as such by writing 1 to all its bits. In the following code, port 2 is configured first as an input port by writing 1s to it. Then data is received from that port and is sent to P1 continuously.

```
        MOV   A,#0FFH    ;A=FF hex
        MOV   P2,A       ;make P2 an input port by
                         ;writing all 1s to it
BACK:   MOV   A,P2       ;get data from P2
        MOV   P1,A       ;send it to Port 1
        SJMP  BACK       ;keep doing that
```

Dual role of port 2

In systems based on the 8751, 89C51, and DS5000, P2 is used as simple I/O. However, in 8031-based systems, port 2 must be used along with P0 to pro-

vide the 16-bit address for the external memory. As shown in Figure 4-1, Port 2 is also designated as A8 - A15, indicating its dual function. Since an 8031 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0 - A7, it is the job of P2 to provide bits A8 - A15 of the address. In other words, when the 8031 is connected to external memory, P2 is used for the upper 8 bits of the 16-bit address, and it cannot be used for I/O. This is discussed in detail in Chapter 14.

From the discussion so far, we conclude that in systems based on 8751, 89C51, or DS5000 microcontrollers, we have three ports, P0, P1, and P2, for I/O operations. This should be enough for most microcontroller applications. That leaves port 3 for interrupts as well as other signals, as we will see next.

Port 3

Port 3 occupies a total of 8 pins, pins 10 through 17. It can be used as input or output. P3 does not need any pull-up resistors, the same as P1 and P2 did not. Although port 3 is configured as an output port upon reset, this is not the way it is most commonly used. Port 3 has the additional function of providing some extremely important signals such as interrupts. Table 4-2 provides these alternate functions of P3. This information applies to both 8051 and 8031 chips.

Table 4-2: Port 3 Alternate Functions

P3 Bit	Function	Pin
P3.0	RxD	10
P3.1	TxD	11
P3.2	$\overline{\text{INT0}}$	12
P3.3	$\overline{\text{INT1}}$	13
P3.4	T0	14
P3.5	T1	15
P3.6	$\overline{\text{WR}}$	16
P3.7	$\overline{\text{RD}}$	17

P3.0 and P3.1 are used for the RxD and TxD serial communications signals. See Chapter 10 to see how they are connected. Bits P3.2 and P3.3 are set aside for external interrupts, and are discussed in Chapter 11. Bits P3.4 and P3.5 are used for timers 0 and 1, and are discussed in Chapter 9 where timers are discussed. Finally, P3.6 and P3.7 are used to provide the $\overline{\text{WR}}$ and $\overline{\text{RD}}$ signals of external memories connected in 8031-based systems. Chapter 14 discusses how they are used in 8031-based systems. In systems based on the 8751, 89C51, or DS5000, pins 3.6 and 3.7 are used for I/O while the rest of the pins in Port 3 are normally used in the alternate function role.

Review Questions

1. A given 8051 chip has a speed of 16 MHz. What is the range of frequency that can be applied to the XTAL1 and XTAL2 pins?
2. A 16-MHz 8051 system has a machine cycle of _____.
3. Which pin is used to inform the 8051 that the on-chip ROM contains the program?
4. There are total of _____ ports in the 8051 and each has _____ bits.
5. True or false. All of the 8051 ports can be used for both input and output.
6. Upon power up, the program counter (PC) has a value of _____.
7. Upon power up, the 8051 fetches the first opcode from ROM address location _____.
8. Which of the 8051 ports need pull-up resistors to function as an I/O port?

SECTION 4.2: I/O PROGRAMMING; BIT MANIPULATION

In this section we further examine 8051 I/O instructions. We pay special attention to I/O bit manipulation since it is a powerful and widely used 8051 feature. A detailed discussion of I/O ports of the 8051 is given in Appendix C.2.

Different ways of accessing the entire 8 bits

In the following code, as in many previous I/O examples, the entire 8 bits of Port 1 are accessed.

```
BACK:   MOV   A,#55H
        MOV   P1,A
        ACALL DELAY
        MOV   A,#0AAH
        MOV   P1,A
        ACALL DELAY
        SJMP  BACK
```

The above code toggles every bit of P1 continuously. We have seen a variation of the above program before. Now we can rewrite the above code in a more efficient manner by accessing the port directly without going through the accumulator. This is shown next.

```
BACK:   MOV   P1,#55H
        ACALL DELAY
        MOV   P1,#0AAH
        ACALL DELAY
        SJMP  BACK
```

We can write another variation of the above code by using a technique called *read-modify-write*. This is shown next.

Read-modify-write feature

The ports in the 8051 can be accessed by the read-modify-write technique. This feature saves many lines of code by combining in a single instruction all three actions of (1) reading the port, (2) modifying it, and (3) writing to the port. The following code first places 01010101 (binary) into port 1. Next, the instruction "XLR P1,#0FFH" performs an XOR logic operation on P1 with 1111 1111 (binary), and then writes the result back into P1.

```
        MOV   P1,#55H          ;P1=01010101
AGAIN:  XLR   P1,#0FFH         ;EX-OR P1 with 1111 1111
        ACALL DELAY
        SJMP  AGAIN
```

Notice that the XOR of 55H and FFH gives AAH. Likewise, the XOR of AAH and FFH gives 55H. Logic instructions are discussed in Chapter 7.

Single-bit addressability of ports

There are times that we need to access only 1 or 2 bits of the port instead of the entire 8 bits. A powerful feature of 8051 I/O ports is their capability to access individual bits of the port without altering the rest of the bits in that port. For example, the following code toggles the bit P1.2 continuously.

```
BACK:     CPL P1.2          ;complement P1.2 only
          ACALL DELAY
          SJMP BACK

;another variation of the above program follows
AGAIN:    SETB P1.2         ;change only P1.2=high
          ACALL DELAY
          CLR  P1.2         ;change only P1.2=low
          ACALL DELAY
          SJMP AGAIN
```

Notice that P1.2 is the third bit of P1, since the first bit is P1.0, the second bit is P1.1, and so on. Table 4-3 shows the bits of 8051 I/O ports. See Example 4-2 for an example of bit manipulation of I/O bits. Notice in Example 4-2 that unused portions of Ports 1 and 2 are undisturbed. This single-bit addressability of I/O ports is one of most powerful features of the 8051 microcontroller.

Table 4-3: Single-Bit Addressability of Ports

P0	P1	P2	P3	Port Bit
P0.0	P1.0	P2.0	P3.0	D0
P0.1	P1.1	P2.1	P3.1	D1
P0.2	P1.2	P2.2	P3.2	D2
P0.3	P1.3	P2.3	P3.3	D3
P0.4	P1.4	P2.4	P3.4	D4
P0.5	P1.5	P2.5	P3.5	D5
P0.6	P1.6	P2.6	P3.6	D6
P0.7	P1.7	P2.7	P3.7	D7

Example 4-2

Write a program to perform the following.
(a) Keep monitoring the P1.2 bit until it becomes high,
(b) When P1.2 becomes high, write value 45H to port 0, and
(c) Send a high-to-low (H-to-L) pulse to P2.3.

Solution:

```
          SETB P1.2          ;make P1.2 an input
          MOV  A,#45H        ;A=45H
AGAIN:    JNB  P1.2,AGAIN    ;get out when P1.2=1
          MOV  P0,A          ;issue A to P0
          SETB P2.3          ;make P2.3 high
          CLR  P2.3          ;make P2.3 low for H-to-L
```

In this program, instruction "JNB P1.2,AGAIN" (JNB means jump if no bit) stays in the loop as long as P1.2 is low. When P1.2 becomes high, it gets out of the loop, writes the value 45H to port 0, and creates a H-to-L pulse by the sequence of instructions SETB and CLR.

Review Questions

1. Upon reset, the 8051 ports are configured as
   (a) input (b) output (c) both input and output.
2. True or false. The instruction "`SETB P2.1`" makes pin P2.1 high while leaving other bits of P2 unchanged.
3. Why do we use 55H and AAH to test the bits of the port?
4. Is the following a valid instruction: "`MOV P1,#99H`"? Explain your answer.
5. Using the instruction "`JNB P2.5,HERE`" assumes that bit P2.5 is an _____ (input, output).

SUMMARY

This chapter began by describing the function of each pin of the 8051. The four ports of the 8051, P0, P1, P2, and P3, each use 8 pins, making them 8-bit ports. These ports can be used for input or output. Port 0 can be used for either address or data. Port 3 can be used to provide interrupt and serial communication signals. Then I/O instructions of the 8051 were explained, and numerous examples were given.

PROBLEMS

SECTION 4.1: PIN DESCRIPTION OF THE 8051

1. The 8051 DIP package is a ____-pin package.
2. Which pins are assigned to V_{CC} and GND?
3. In the 8051, how many pins are designated as I/O port pins?
4. The crystal oscillator is connected to pins ____ and ____ .
5. If an 8051 is rated as 25 MHz, what is the maximum frequency that can be connected to it?
6. Indicate the pin number assigned to RST in the DIP package.
7. RST is an _____ (input, output) pin.
8. The RST pin is normally _____ (low, high) and needs a _____ (low, high) signal to be activated.
9. What are the contents of the PC (program counter) upon RESET of the 8051?
10. What are the contents of the SP register upon RESET of the 8051?
11. What are the contents of the A register upon RESET of the 8051?
12. Find the machine cycle for the following crystal frequencies connected to X1 and X2.
    (a) 12 MHz (b) 20 MHz (c) 25 MHz (d) 30 MHz
13. $\overline{EA}$ stands for _____ and is an _____ (input, output) pin.
14. For 8051 family members with on-chip ROM such as the 8751 and the 89C51, pin $\overline{EA}$ is connected to _____ (V_{CC}, GND).
15. $\overline{PSEN}$ is an _____ (input, output) pin.
16. ALE is an _____ (input, output) pin.
17. ALE is used mainly in systems based on the _____ (8051, 8031).
18. How many pins are designated as P0 and what are those in the DIP package?

19. How many pins are designated as P1 and what are those in the DIP package?
20. How many pins are designated as P2 and what are those in the DIP package?
21. How many pins are designated as P3 and what are those in the DIP package?
22. Upon RESET, all the bits of ports are configured as _____ (input, output).
23. In the 8051, which port needs a pull-up resistor to be used as I/O?
24. Which port of the 8051 does not have any alternate function and can be used solely for I/O?
25. Write a program to get 8-bit data from P1 and send it to ports P0, P2, and P3.
26. Write a program to get an 8-bit data from P2 and send it to ports P0 and P1.
27. In P3, which pins are for RxD and TxD?
28. At what memory location does the 8051 wake up upon RESET? What is the implication of that?
29. Write a program to toggle all the bits of P1 and P2 continuously
    (a) using AAH and 55H (b) using the CPL instruction.
30. What is the address of the last location of on-chip ROM for the 8751?

SECTION 4.2: I/O PROGRAMMING; BIT MANIPULATION

31. Which ports of the 8051 are bit-addressable?
32. What is the advantage of bit-addressability for 8051 ports?
33. When P1 is accessed as a single bit port, it is designated as _____.
34. Is the instruction "CPL P1" a valid instruction?
35. Write a program to toggle P1.2 and P1.5 continuously without disturbing the rest of the bits.
36. Write a program to toggle P1.3, P1.7, and P2.5 continuously without disturbing the rest of the bits.
37. Write a program to monitor bit P1.3. When it is high, send 55H to P2.
38. Write a program to monitor the P2.7 bit. When it is low, send 55H and AAH to P0 continuously.
39. Write a program to monitor the P2.0 bit. When it is high, send 99H to P1.
40. Write a program to monitor the P1.5 bit. When it is high, make a low-to-high-to-low pulse on P1.3.

ANSWERS TO REVIEW QUESTIONS

SECTION 4.1: PIN DESCRIPTION OF THE 8051

1. From 0 to 16 MHz, but no more than 16 MHz.
2. 1/12th of 16 MHz is 1.33 MHz. and the machine cycle is = 0.75 μs
3. EA
4. 4, 8
5. True
6. PC = 0000
7. 0000
8. Port 0

SECTION 4.2: I/O PROGRAMMING; BIT MANIPULATION

1. (b)
2. True
3. They are the complement of each other.
4. Yes. This is called immediate addressing mode (discussed in Chapter 5).
5. input