CHAPTER 2

8051 ASSEMBLY LANGUAGE PROGRAMMING

OBJECTIVES

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

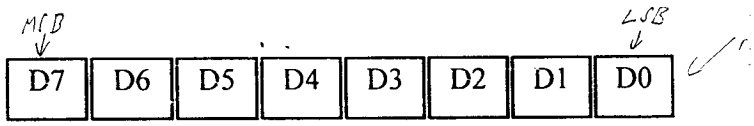
- **List the registers of the 8051 microcontroller**
- **Manipulate data using the registers and MOV instructions**
- **Code simple 8051 Assembly language instructions**
- **Assemble and run an 8051 program**
- **Describe the sequence of events that occur upon 8051 power-up**
- **Examine programs in 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.

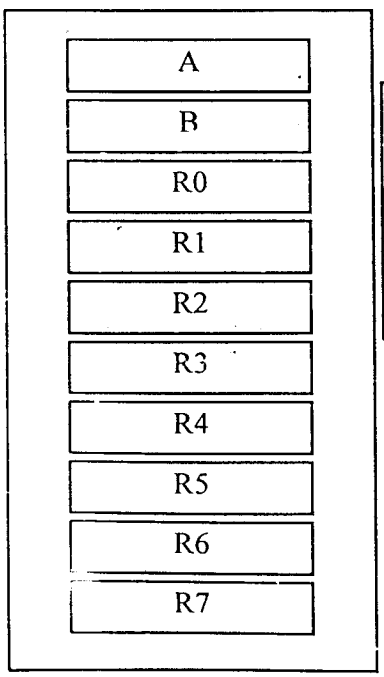


Figure 2-1 (a): Some 8-bit Registers of the 8051

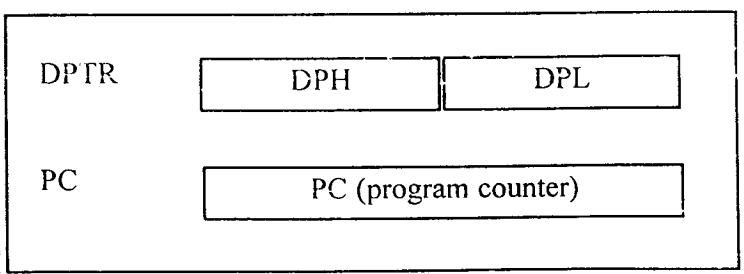


Figure 2-1 (b): Some 8051 16-bit Registers

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

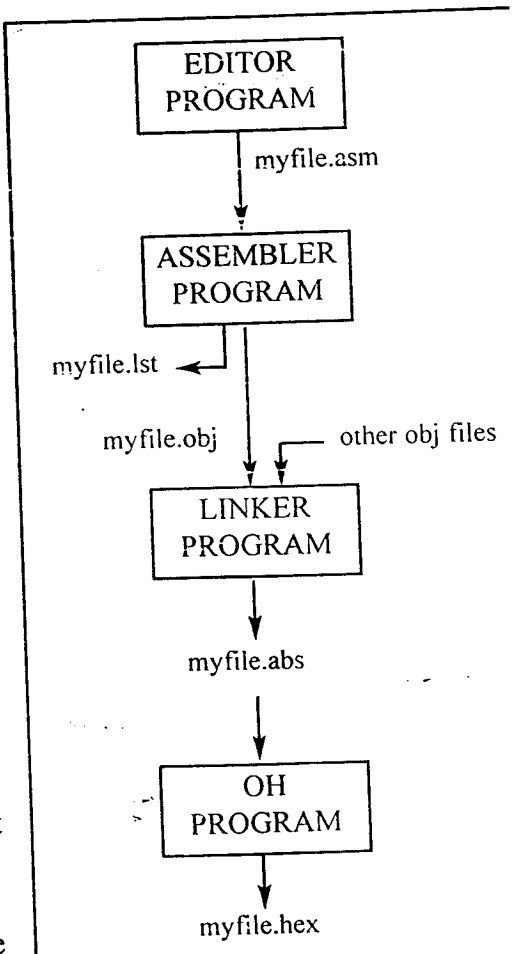


Figure 2-2. Steps to Create a Program

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.

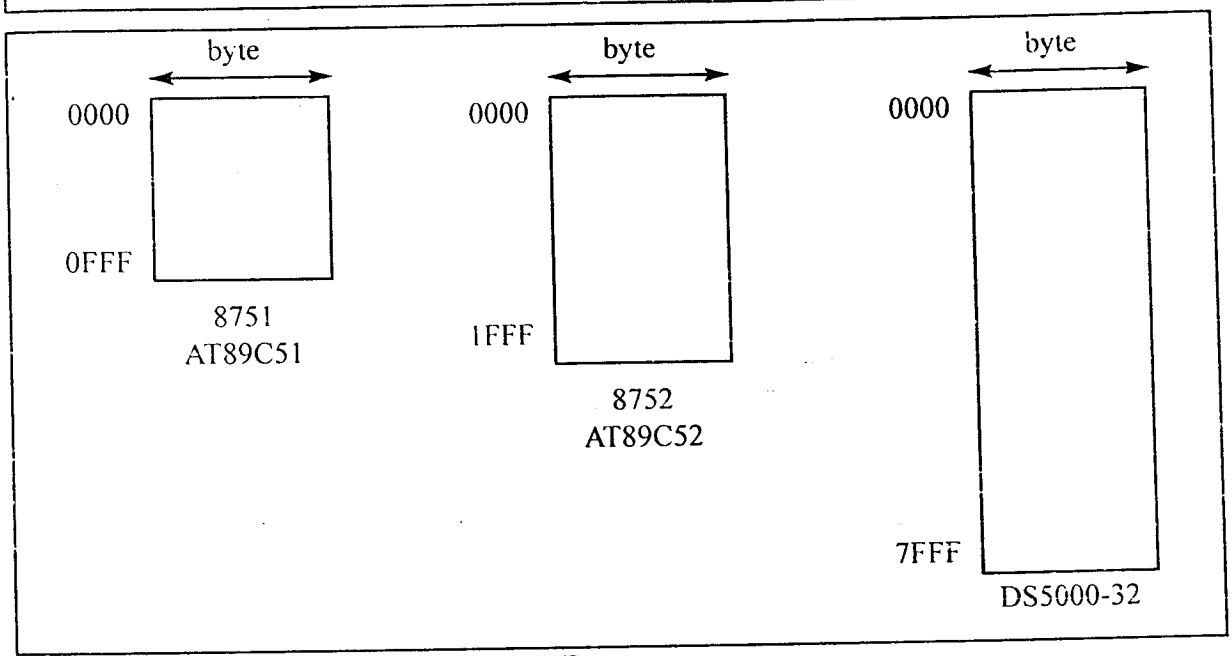


Figure 2-3. 8051 On-Chip ROM Address Range

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.

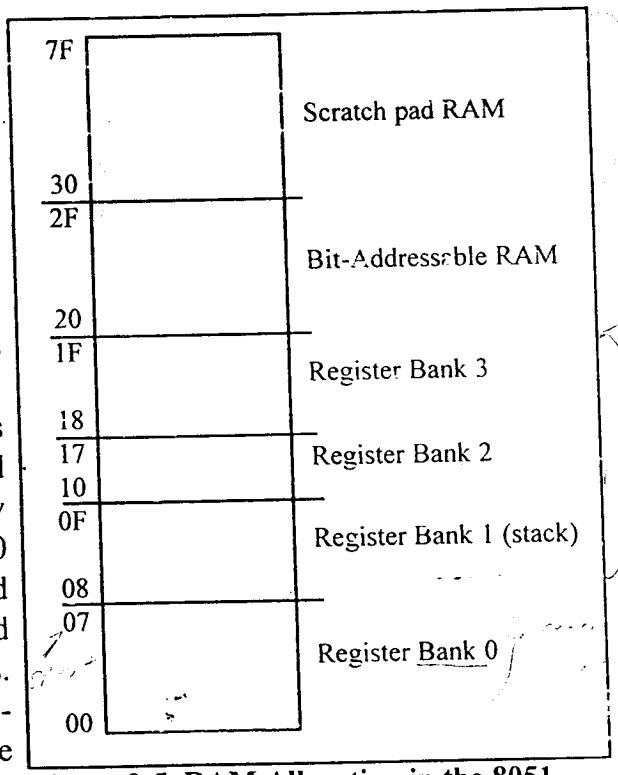


Figure 2-5. RAM Allocation in the 8051

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:

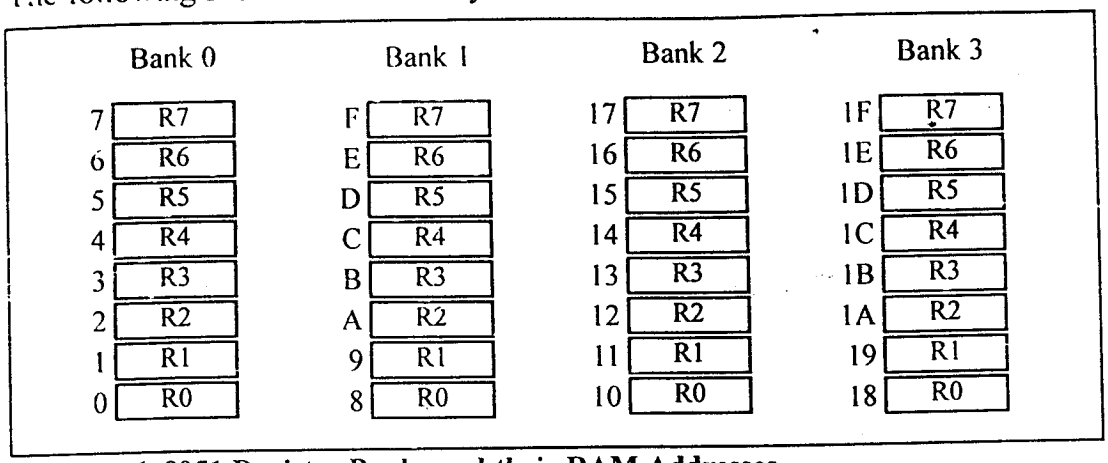


Figure 2-6. 8051 Register Banks and their RAM Addresses

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

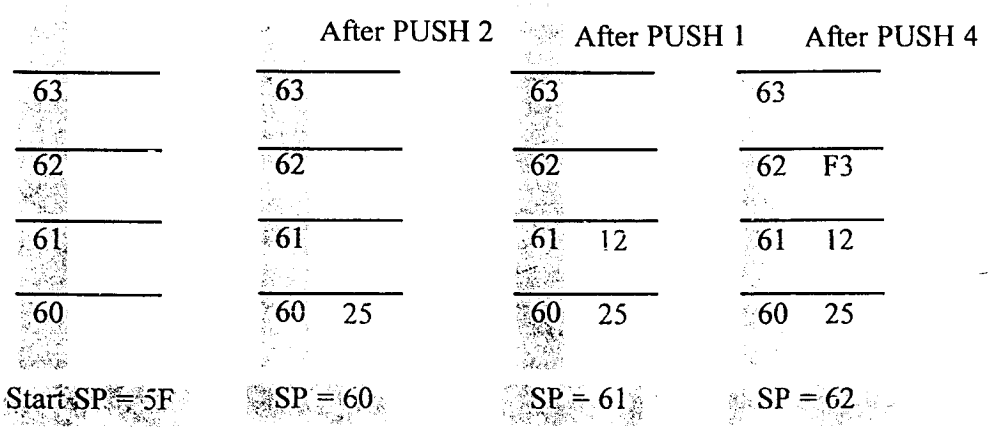
In addition to using the stack to save registers, the CPU also uses the stack to save the address of the instruction just below the CALL instruction. This is how the CPU knows where to resume when it returns from the called subroutine. More information on this will be given in Chapter 3 when we discuss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