MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI

PG & RESEARCH DEPARTMENT OF PHYSICS

E-NOTES

CLASS : II MSC PHYSICS

SUBJECT: MICROPROCESSOR (8085) & MICROCONTROLLER (8051)

SUBJECT CODE: GPH33

SYLLABUS

Unit-3: Basic of Microcontroller 8051

8051 Micro-controller hardware: 8051 oscillator and clock - Program counter and data pointer - A and B CPU register - Flags and PSW - Internal memory - Internal RAM - Stack and stack pointer - Special function registers - Internal ROM-Input / output pin, ports and circuits - External memory.

Counter and Timer: Counter / Timer interrupts - Timing - Timer modes of operation – Counting-Serial data input / Output: Serial data interrupt - Data transmission - Data reception - serial data transmission modes.

for reducing programmer time (there is *always* a price) is the size of the program generated.

1.3 Four-Bit to Thirty-Two-Bit Microcontrollers

Every application demands a microcontroller that offers the right amount of functionality at the minimum cost. Applications vary from controlling an appliance to controlling an automobile. No single microcontroller design can economically meet these demands, so semiconductor manufacturers offer an array of microcontrollers designed to handle data in 4-, 8-, 16- and 32-bit words.

Bits are not the only thing that increase as word length grows. Additional functions are also added to meet market needs. Some of the more popular additional functions are

- Analog-to-digital (A/D) converters, which change external analog signals to digital bits
- Counter arrays, used to count and generate pulses
- Watchdog timers (WDTs), which reset the controller if program execution hangs up
- Serial data, both asynchronous (UART) and synchronous
- Pulse width modulation (PWM), used, among other things, for motor speed control
- Phase-locked-loops (PLLs), used for synchronous communications
- External bus controllers for static (RAM/ROM) and dynamic (SDRAM) memories

If the market is large enough, say for automotive dashboard control and display, a high-end microcontroller will be designed and marketed by numerous companies just to meet that single application.

Four-Bit Microcontrollers

In terms of sheer volume, 4-bit microcontrollers may be used more than any other type. For a commodity microcontroller, cost depends as much on the volume of the package and the number of pins as on the amount of silicon inside. Pin count, in turn, depends on the number of data bits commonly handled by the microcontroller, and its I/O capability. Using 4 bits reduces package cost and pin count to a minimum. Employed in applications ranging from alphanumeric LED/LCD display drivers to portable battery chargers, these are the least expensive "smart chips" available. An example of a contemporary 4-bit microcontroller is the Renasas M34501, in a 20-pin DIP:

4-bit RAM	10-bit ROM	Counters	I/O Pins	Other Features
256	4K	2	14	A/D, WDT

Eight-Bit Microcontrollers

Eight bit microcontrollers are perhaps the most popular microcontrollers in use today, judging from the number of semiconductor companies making them.

Eight bits have proven to be a very useful word size for everyday controller tasks. Capable of 256 decimal values, or quarter-percent resolution, the 1-byte data word is adequate for many control and monitoring applications. Serial ASCII code is also byte size, making 8 bits the natural choice for data communication applications. In addition, most low-cost RAM and ROM memories store 1 byte per memory location for easy interfacing to an 8-bit microcontroller.

One indication of the popularity of 8-bit microcontrollers is the fact that some 44 manufacturers produce over 600 models based on the 8051 architecture alone! Other popular microcontrollers, such as those designed by Microchip, Motorola, and Zilog, add hundreds of additional choices to the 8-bit menu. Here is what the 8051 offers when mounted in a 40-pin DIP:

ROM	ROM	Counters	I/O Pins	Other Features
128 bytes	4K	2	32	UART

Variations on a design occur when manufacturers offer models that include application-specific extras such as A/D and D/A converters, counter arrays, UARTs, WDT, and different memory configurations. On-chip ROM memory size may be increased, or ROMless models made that use off chip EPROMs for prototyping purposes. Flash and EEPROM memories may also be incorporated into the design.

Sixteen-Bit Microcontrollers

Sixteen-bit microcontrollers offer much of the generality of 8-bit models, but with greatly increased memory size and speed. Sixteen-bit microcontrollers are also much better suited for programming in high-level languages, such as C.

Applications for 16-bit microcontrollers are calculation and data intensive and include disk drives, modems, printers, scanners, pattern recognition, and automotive and servomotor control. A typical 16-bit microcontroller, the Motorola 68HC16Z3, has these attributes when mounted in a 144-lead LQFP package:

RAM	ROM	Counters	I/O Pins	Other Features
4K	8K	2	24	Counter Array, UART, A/D, WDT

Thirty-Two-Bit Microcontrollers

Thirty-two-bit microcontrollers are currently evolving away from general-purpose applications to targeted markets such as PDAs, GPS, automotive control, communication networks, robotics, entertainment/game boxes, digital cameras, cell phones, and similar high-end uses. As an example, the Sharp LH79520 housed in a 176-pin LQFP package offers the following features, most of which could be used to implement a notebook computer: 32K RAM, color LCD controller, three UARTs, synchronous serial, two PWMs, 64 I/O pins, four counters, WDT, real-time clock, PLL, and direct memory access (DMA). Clearly, the development of 32-bit microcontrollers is driven by large, well-defined markets, and must be studied on an individual basis.

4 Development Systems for Microcontrollers

What is needed to be able to apply a microcontroller to your product? That is, what package of hardware and software will allow the microcontroller to be programmed and connected to your application? A package commonly called a *development system* is required.

First, trained personnel must be available either on your technical staff or as consultants. One person who is versed in digital hardware and computer software is the minimum number.

Second, a device capable of programming EPROMs must be available to test the prototype device. Many of the microcontroller families discussed have a ROMless version, an EPROM version, or an electrically erasable and programmable read only memory (EEPROM) version that lets the designer debug the hardware and software prototype before committing to full-scale production. Many inexpensive EPROM programmers are sold that plug into a port of most popular personal computers. More expensive, and more versatile, dedicated programmers are also available. An alternative to EPROMs are vendor-supplied prototype cards that allow code to be downloaded from a host computer and the program run from RAM for debugging purposes. An EPROM will eventually have to be programmed for the production version of the microcontroller.

Finally, software is needed, along with a personal computer to host it. The minimum software package consists of a machine-language assembler, which can be supplied by the microcontroller vendor or bought from independent de-

OBJECTIVES

On successful completion of this chapter you will be able to:

- Describe the hardware features of the 8051 microcontroller.
- List the internal registers of the 8051 microcontroller and their functions.
- Draw the machine cycle for the 8051 microcontroller.
- State the physical differences between the Port 0, 1, 2, and 3 I/O pins.
- Describe the various operating modes of the timer/counters and associated control registers.
- Describe the various operating modes of the UART, and associated control registers.
- List the types of interrupts, the interrupt program addresses, and the interrupt control registers.

0 Introduction

The first task faced when learning to use a new computer is to become familiar with the capability of the machine. The features of the computer are best learned by studying the internal hardware design, also called the architecture of the device, to determine the type, number, and size of the registers and other circuitry.

The hardware is manipulated by an accompanying set of program instructions, or software, which is usually studied next. Once familiar with the hardware and software, the system designer can then apply the microcontroller to the problems at hand.

A natural question during this process is "What do I do with all this stuff?" Similar to attempting to write a poem in a foreign language before you have a vocabulary and rules of grammar, writing meaningful programs is not possible until you have become acquainted with both the hardware and the software of a computer.

This chapter provides a broad overview of the architecture of the 8051. In subsequent chapters, we will cover in greater detail the interaction between the hardware and the software.

1 8051 Microcontroller Hardware

The 8051 microcontroller generic part number actually includes a whole family of microcontrollers that have numbers ranging from 8031 to 8751 and are available in N-Channel Metal Oxide Silicon (NMOS) and Complementary Metal Oxide

architecture. This galaxy of parts, the result of desires by the manufacturers to leave no market niche unfilled, would require many chapters to cover. In this chapter, we will study a "generic" 8051, housed in a 40-pin DIP, and direct the investigation of a particular type to the data books. The block diagram of the 8051 in Figure 3.1a shows all of the features unique to microcontrollers:

Internal ROM and RAM

I/O ports with programmable pins

Timers and counters

Serial data communication

The figure also shows the usual CPU components: program counter, ALU, working registers, and clock circuits.[1]

The 8051 architecture consists of these specific features:

- Eight-bit CPU with registers A (the accumulator) and B
- Sixteen-bit program counter (PC) and data pointer (DPTR)
- Eight-bit program status word (PSW)
- Eight-bit stack pointer (SP)
- Internal ROM or EPROM (8751) of 0 (8031) to 4K (8051)
- Internal RAM of 128 bytes:
  - Four register banks, each containing eight registers
  - Sixteen bytes, which may be addressed at the bit level
  - Eighty bytes of general-purpose data memory
- Thirty-two input/output pins arranged as four 8-bit ports: P0 – P3
- Two 16-bit timer/counters: T0 and T1
- Full duplex serial data receiver/transmitter: SBUF
- Control registers: TCON, TMOD, SCON, PCON, IP, and IE
- Two external and three internal interrupt sources
- Oscillator and clock circuits

[1] Knowledge of the details of circuit operation that cannot be affected by any instruction or external data, although intellectually stimulating, tends to confuse the student new to the 8051. For this reason, this text concentrates on the essential features of the 8051; the more advanced student may wish to refer to manufacturers' data books for additional information.

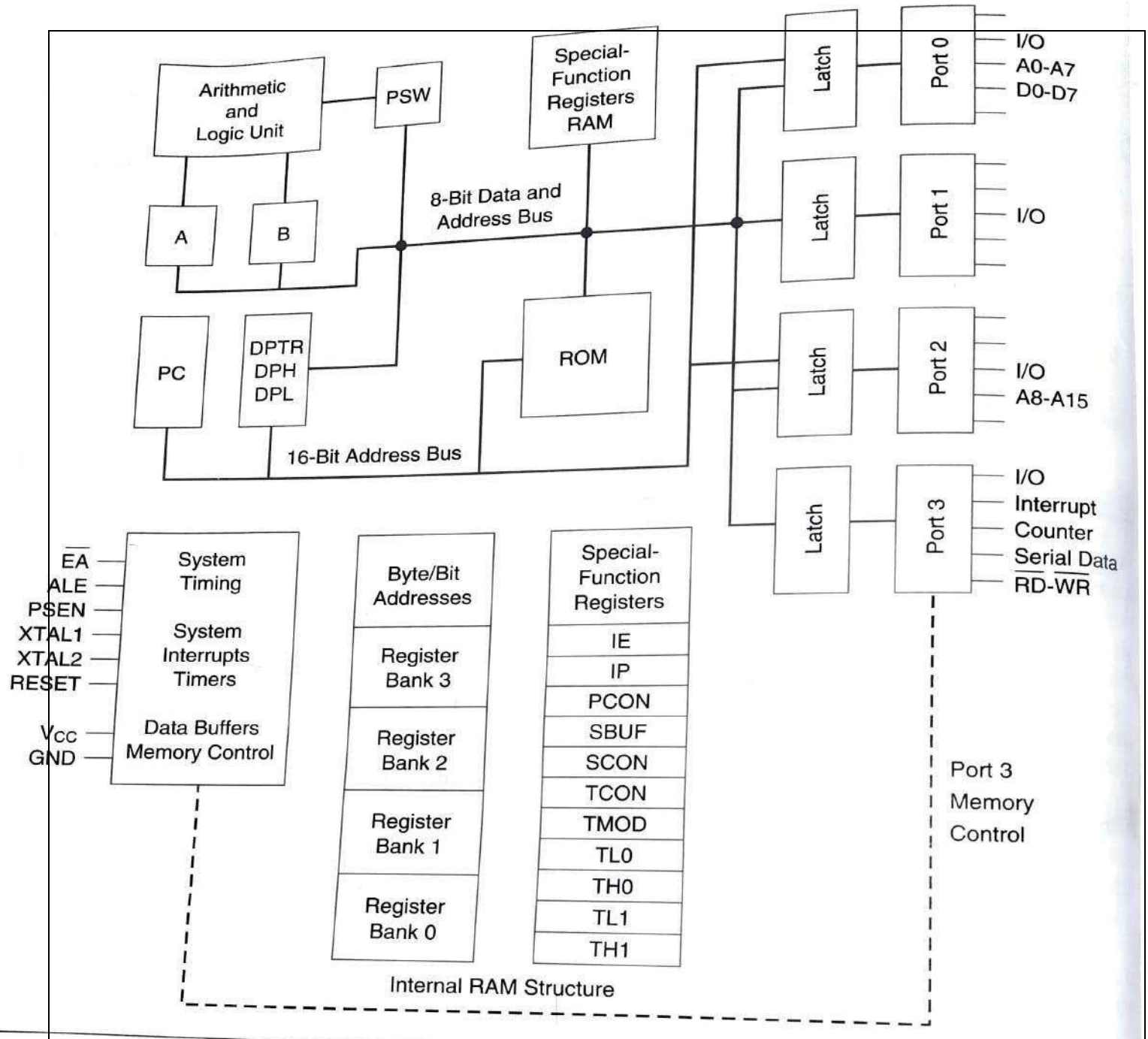


FIGURE 3.1A ◆ 8051 Block Diagram

The programming model of the 8051 in Figure 3.1b shows the 8051 as a collection of 8- and 16-bit registers and 8-bit memory locations. These registers and memory locations can be made to operate using the software instructions that are incorporated as part of the design. The program instructions have to do with the control of the registers and digital data paths that are physically con-

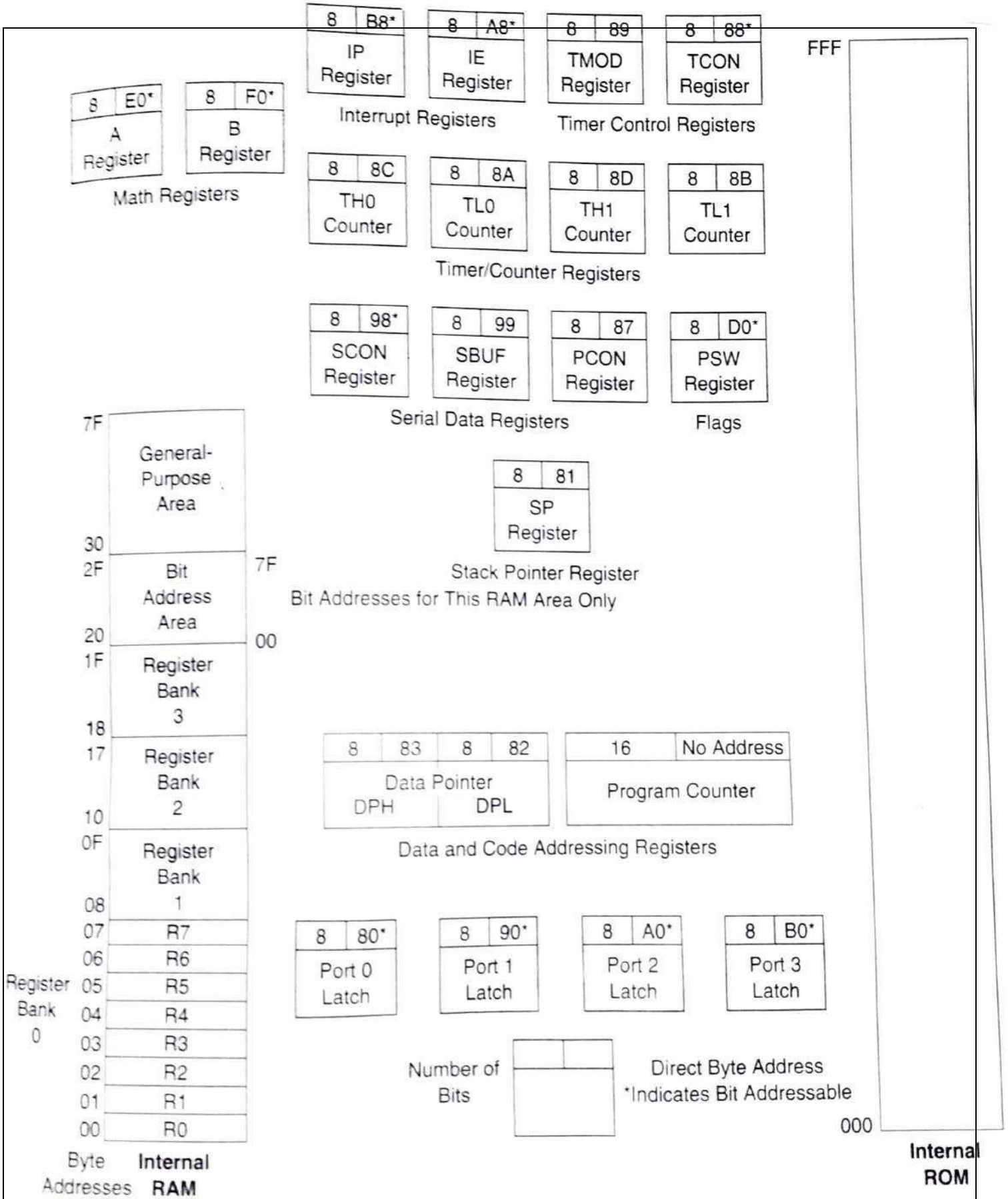


FIGURE 3.1B ◆ 8051 Programming Model

tained inside the 8051, as well as memory locations that are physically located outside the 8051.

The model is complicated by the number of special-purpose registers that must be present to make a microcomputer a microcontroller. A cursory inspection of the model is recommended for the first-time viewer; return to the model as needed while progressing through the remainder of the text.

Most of the registers have a specific function; those that do occupy an individual block with a symbolic name, such as A or TH0 or PC. Others, which are generally indistinguishable from each other, are grouped in a larger block, such as internal ROM or RAM memory.

Each register, with the exception of the program counter, has an internal 1-byte address assigned to it. Some registers (marked with an asterisk* in Figure 3.1b) are both byte and bit addressable. That is, the entire byte of data at such register addresses may be read or altered, or individual bits may be read or altered. Software instructions are generally able to specify a register by its address, its symbolic name, or both.

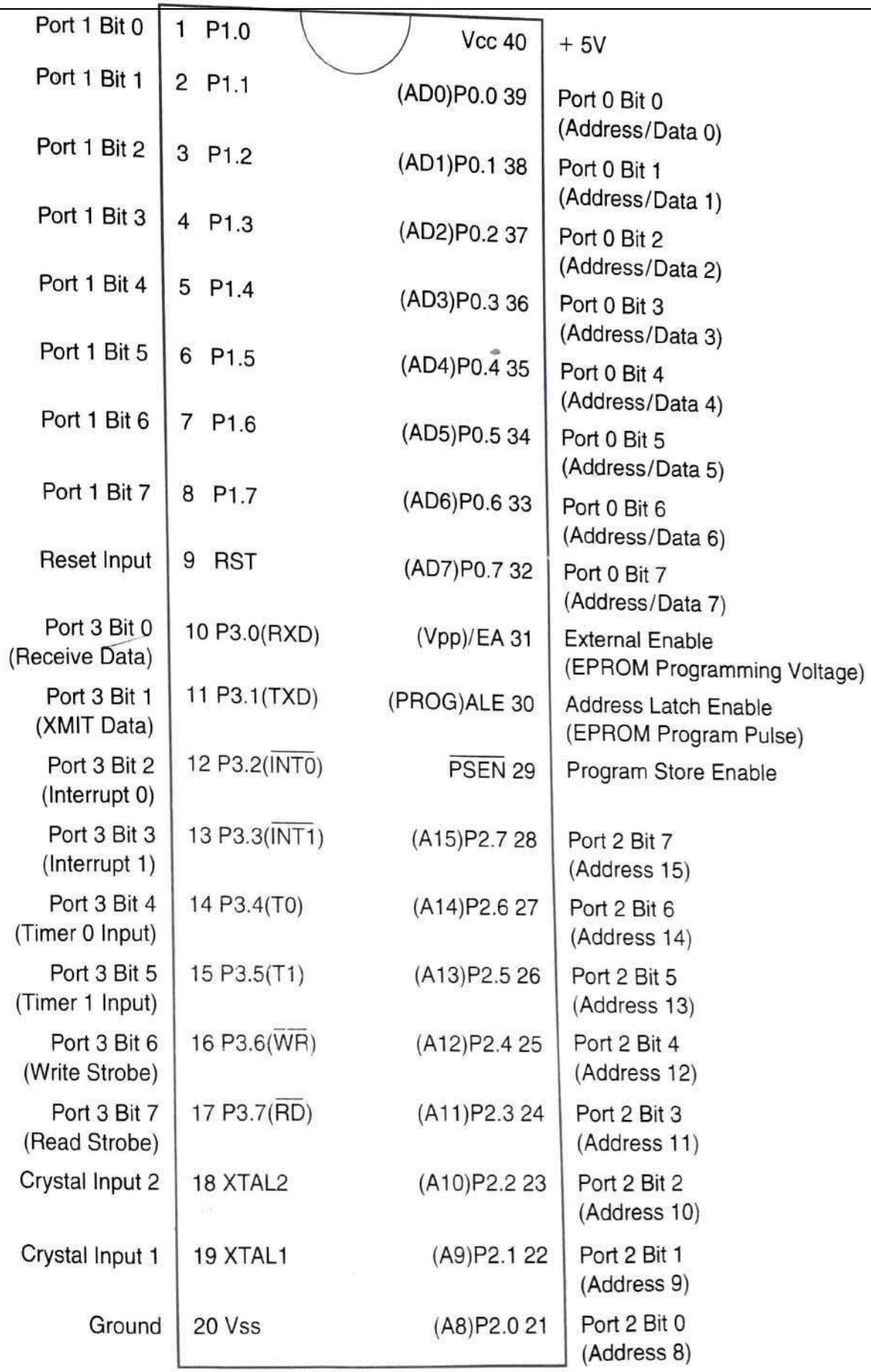
A pinout of the 8051 packaged in a 40-pin DIP is shown in Figure 3.2 with the full and abbreviated names of the signals for each pin. It is important to note that many of the pins are used for more than one function (the alternate functions are shown in parentheses in Figure 3.2). Not all of the possible 8051 features may be used *at the same time.*

Programming instructions or physical pin connections determine the use of any multifunction pins. For example, port 3 bit 0 (abbreviated P3.0) may be used as a general-purpose I/O pin, or as an input (RXD) to SBUF, the serial data receiver register. The system designer decides which of these two functions is to be used and designs the hardware and software affecting that pin accordingly.

The 8051 Oscillator and Clock

The heart of the 8051 is the circuitry that generates the clock pulses by which all internal operations are synchronized. Pins XTAL1 and XTAL2 are provided for connecting a resonant network to form an oscillator. Typically, a quartz crystal and capacitors are employed, as shown in Figure 3.3. The crystal frequency is the basic internal clock frequency of the microcontroller. The manufacturers make available 8051 designs that can run at specified maximum and *minimum* frequencies, typically 1 megahertz to 16 megahertz. Minimum frequencies imply that some internal memories are dynamic and must always operate above a minimum frequency or data will be lost.

Serial data communication needs often dictate the frequency of the oscillator because of the requirement that internal counters must divide the basic clock rate to yield standard communication bit per second (baud) rates. If the basic clock frequency is not divisible without a remainder, then the resulting communication frequency is not standard.



Note: Alternate functions are shown below the port name (in parentheses). Pin numbers and pin names are shown inside the DIP package.

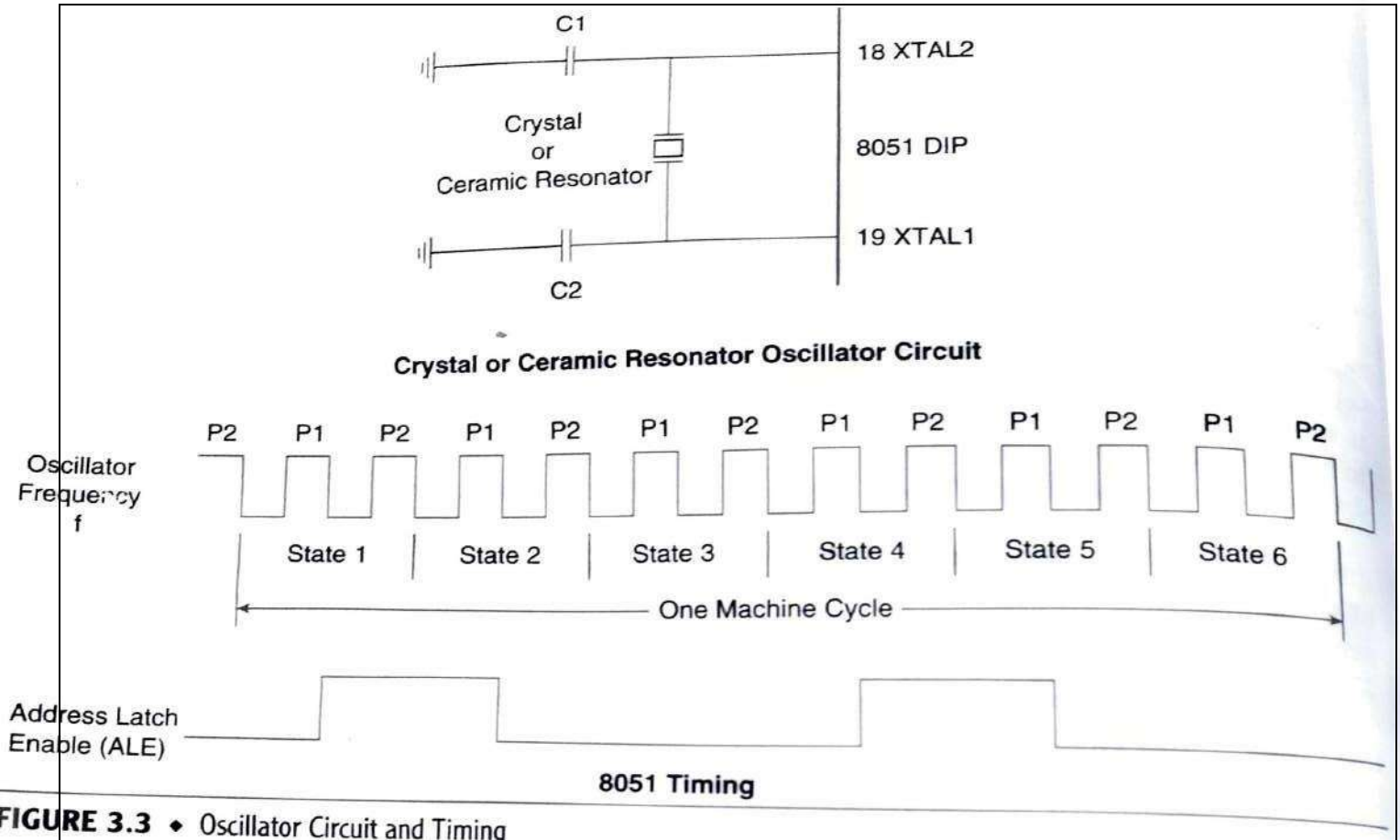


FIGURE 3.3 ◆ Oscillator Circuit and Timing

Ceramic resonators may be used as a low-cost alternative to crystal resonators. However, decreases in frequency stability and accuracy make the ceramic resonator a poor choice if high-speed serial data communication with other systems, or critical timing, is to be done.

The oscillator formed by the crystal, capacitors, and an on-chip inverter generates a pulse train at the frequency of the crystal, as shown in Figure 3.3.

The clock frequency, *f*, establishes the smallest interval of time within the microcontroller, called the pulse, *P*, time. The smallest interval of time to accomplish any simple instruction, or part of a complex instruction, however, is the machine cycle. The machine cycle is itself made up of six states. A state is the basic time interval for discrete operations of the microcontroller such as fetching an opcode byte, decoding an opcode, executing an opcode, or writing a data byte. Two oscillator pulses define each state.

Program instructions may require one, two, or four machine cycles to be executed, depending on the type of instruction. Instructions are fetched and executed by the microcontroller automatically, beginning with the instruction

crystal, although seemingly an odd value, yields a cycle frequency of 921.6 kilohertz, which can be divided evenly by the standard communication baud rates of 19200, 9600, 4800, 2400, 1200, and 300 hertz.

Note, in Figure 3.3, there are two ALE pulses per machine cycle. The ALE pulse, which is primarily used as a timing pulse for external memory access, indicates when every instruction byte is fetched. Two bytes of a single instruction may thus be fetched, and executed, in one machine cycle. Single-byte instructions are not executed in a half cycle, however. Single-byte instructions "throw-away" the second byte (which is the first byte of the next instruction). The next instruction is then fetched in the following cycle.

Program Counter and Data Pointer

The 8051 contains two 16-bit registers: the program counter (PC) and the data pointer (DPTR). Each is used to hold the address of a byte in memory.

Program instruction bytes are fetched from locations in memory that are addressed by the PC. Program ROM may be on the chip at addresses 0000h to 0FFFh, external to the chip for addresses that exceed 0FFFh, or totally external for all addresses from 0000h to FFFFh. The PC is automatically incremented after every instruction byte is fetched and may also be altered by certain instructions. The PC is the only register that does not have an internal address.

The DPTR register is made up of two 8-bit registers, named DPH and DPL, which are used to furnish memory addresses for internal and external code access and external data access. The DPTR is under the control of program instructions and can be specified by its 16-bit name, DPTR, or by each individual byte name, DPH and DPL. DPTR does not have a single internal address; DPH and DPL are each assigned an address.

A and B CPU Registers

The 8051 contains 34 general-purpose, or working, registers. Two of these, registers A and B, hold results of many instructions, particularly math and logical

operations, of the 8051 central processing unit (CPU). The other 32 are arranged as part of internal RAM in four banks, B0 – B3, of eight registers and comprise the mathematical core.

The A (accumulator) register is the most versatile of the two CPU registers and is used for many operations, including addition, subtraction, integer multiplication and division, and Boolean bit manipulations. The A register is also used for all data transfers between the 8051 and any external memory. The B register is used with the A register for multiplication and division operations and has no other function other than as a location where data may be stored.

Flags and the Program Status Word (PSW)

Flags are 1-bit registers provided to store the results of certain program instructions. Other instructions can test the condition of the flags and make decisions based on the flag states. In order that the flags may be conveniently addressed, they are grouped inside the program status word (PSW) and the power control (PCON) registers.

The 8051 has four math flags that respond automatically to the outcomes of math operations and three general-purpose user flags that can be set to 1 or cleared to 0 by the programmer as desired. The math flags include Carry (C), Auxiliary Carry (AC), Overflow (OV), and Parity (P). User flags are named F0, GF0, and GF1; they are general-purpose flags that may be used by the programmer to record some event in the program. Note that all of the flags can be set and cleared by the programmer at will. The math flags, however, are also affected by math operations.

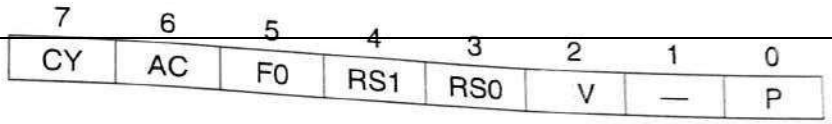
The program status word is shown in Figure 3.4. The PSW contains the math flags, user program flag F0, and the register select bits that identify which of the four general-purpose register banks is currently in use by the program. The remaining two user flags, GF0 and GF1, are stored in PCON, which is shown in Figure 3.13.

Detailed descriptions of the math flag operations will be discussed in chapters that cover the opcodes that affect the flags. The user flags can be set or cleared using data move instructions covered in Chapter 5.

Internal Memory

A functioning computer must have memory for program code bytes, commonly in ROM, and RAM memory for variable data that can be altered as the program runs. The 8051 has internal RAM and ROM memory for these functions. Additional memory can be added externally using suitable circuits.

Unlike microcontrollers with Von Neumann architectures, which can use a *single* memory address for either program code or data, *but not for both*, the 8051 has a Harvard architecture, which uses the *same address*, in *different* memo-



The Program Status Word (PSW) Special Function Register

Bit	Symbol	Function		
7	CY	Carry flag; used in arithmetic, jump, rotate, and Boolean instructions		
6	AC	Auxiliary Carry flag; used for BCD arithmetic		
5	F0	User flag 0		
4	RS1	Register bank select bit 1		
3	RS0	Register bank select bit 0		
		RS1	RS0	
		0	0	Select register bank 0
		0	1	Select register bank 1
		1	0	Select register bank 2
		1	1	Select register bank 3
2	OV	Overflow flag; used in arithmetic instructions		
1	—	Reserved for future use		
0	P	Parity flag; shows parity of register A: 1 = Odd Parity		
		Bit addressable as PSW.0 to PSW.7		

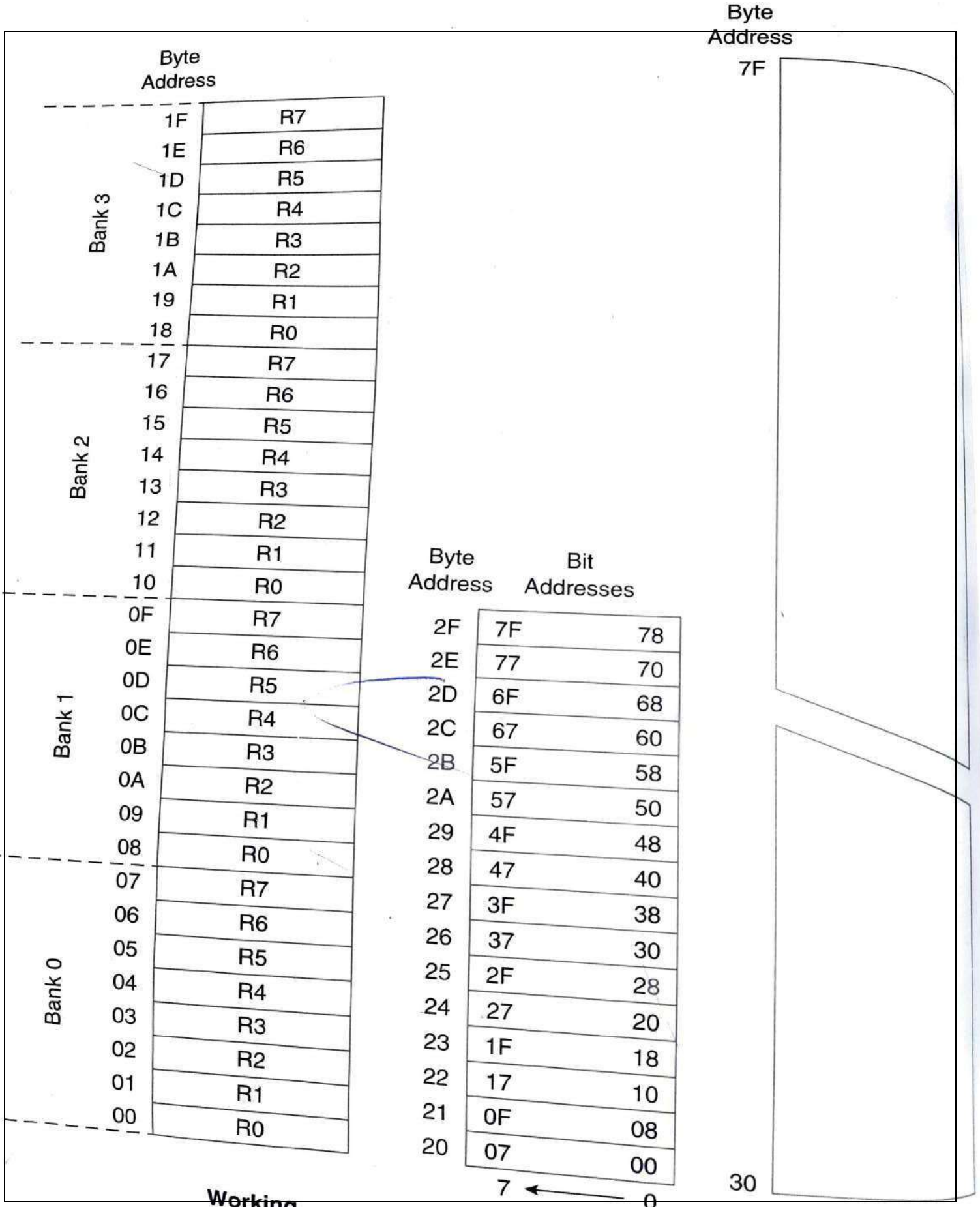
FIGURE 3.4 ◆ PSW Program Status Word Register

ries, for code and data. Internal circuitry accesses the correct memory based on the nature of the operation in progress.

Internal RAM

The 128-byte internal RAM, which is shown generally in Figure 3.1 and in detail in Figure 3.5, is organized into three distinct areas:

1. Thirty-two bytes from address 00h to 1Fh that make up 32 working registers organized as four banks of eight registers each. The four register banks are numbered 0 to 3 and are made up of eight registers named R0 to R7. Each register can be addressed by name (when its bank is selected) or by its RAM address. Thus R0 of bank 3 is R0 (if bank 3 is currently selected) or address 18h (whether bank 3 is selected or not). Bits RS0 and RS1 in the PSW determine which bank of registers is currently in use at any time when the program is running. Register banks not selected can be used as general-purpose RAM. Bank 0 is selected on reset.
2. A *bit*-addressable area of 16 bytes occupies RAM *byte* addresses 20h to 2Fh, forming a total of 128 addressable bits. An addressable bit may be


Byte Address
Bank 3
1F R7
1E R6
1D R5
1C R4
1B R3
1A R2
19 R1
18 R0
Bank 2
17 R7
16 R6
15 R5
14 R4
13 R3
12 R2
11 R1
10 R0
Bank 1
0F R7
0E R6
0D R5
0C R4
0B R3
0A R2
09 R1
08 R0
Bank 0
07 R7
06 R6
05 R5
04 R4
03 R3
02 R2
01 R1
00 R0
Working Registers
Byte Address
Bit Addresses
2F 7F 78
2E 77 70
2D 6F 68
2C 67 60
2B 5F 58
2A 57 50
29 4F 48
28 47 40
27 3F 38
26 37 30
25 2F 28
24 27 20
23 1F 18
22 17 10
21 0F 08
20 07 00
7 ← 0
Bit Addressable
Byte Address
7F
30

specified by its *bit* address of 00h to 7Fh, or 8 bits may form any *byte* address from 20h to 2Fh. Thus, for example, bit address 4Fh is also bit 7 of byte address 29h. Addressable bits are useful when the program need only remember a binary event (switch on, light off, etc.). Internal RAM is in short supply as it is, so why use a byte when a bit will do?

3. A general-purpose RAM area above the bit area, from 30h to 7Fh, addressable as bytes.

The Stack and the Stack Pointer

The stack refers to an area of internal RAM that is used in conjunction with certain opcodes to store and retrieve data quickly. The 8-bit Stack Pointer (SP) register is used by the 8051 to hold an internal RAM address that is called the *top of the stack*. The address held in the SP register is the location in internal RAM where the last byte of data was stored by a stack operation.

When data is to be placed on the stack, the SP increments *before* storing data on the stack so that the stack grows *up* as data is stored. As data is retrieved from the stack, the byte is read from the stack, and then the SP decrements to point to the next available byte of stored data.

Operation of the stack and the SP is shown in Figure 3.6. The SP is set to 07h when the 8051 is reset and can be changed to any internal RAM address by the programmer, using a data move command from Chapter 5.

The stack is limited in height to the size of the internal RAM. The stack has the potential (if the programmer is not careful to limit its growth) to overwrite valuable data in the register banks, bit-addressable RAM, and scratch-pad RAM areas. The programmer is responsible for making sure the stack does not grow beyond predefined bounds!

The stack is normally placed high in internal RAM, by an appropriate choice of the number placed in the SP register, to avoid conflict with the register, bit, and scratch-pad internal RAM areas.

Special Function Registers

The 8051 operations that do not use the internal 128-byte RAM addresses from 00h to 7Fh are done by a group of specific internal registers, each called a Special-Function register (SFR), which may be addressed much like internal RAM, using addresses from 80h to FFh.

Some SFRs (marked with an asterisk* in Figure 3.1b) are also bit addressable, as is the case for the bit area of RAM. This feature allows the programmer to change only what needs to be altered, leaving the remaining bits in that SFR unchanged.

Not all of the addresses from 80h to FFh are used for SFRs, and attempting to use an address that is not defined, or *empty*, results in unpredictable results.

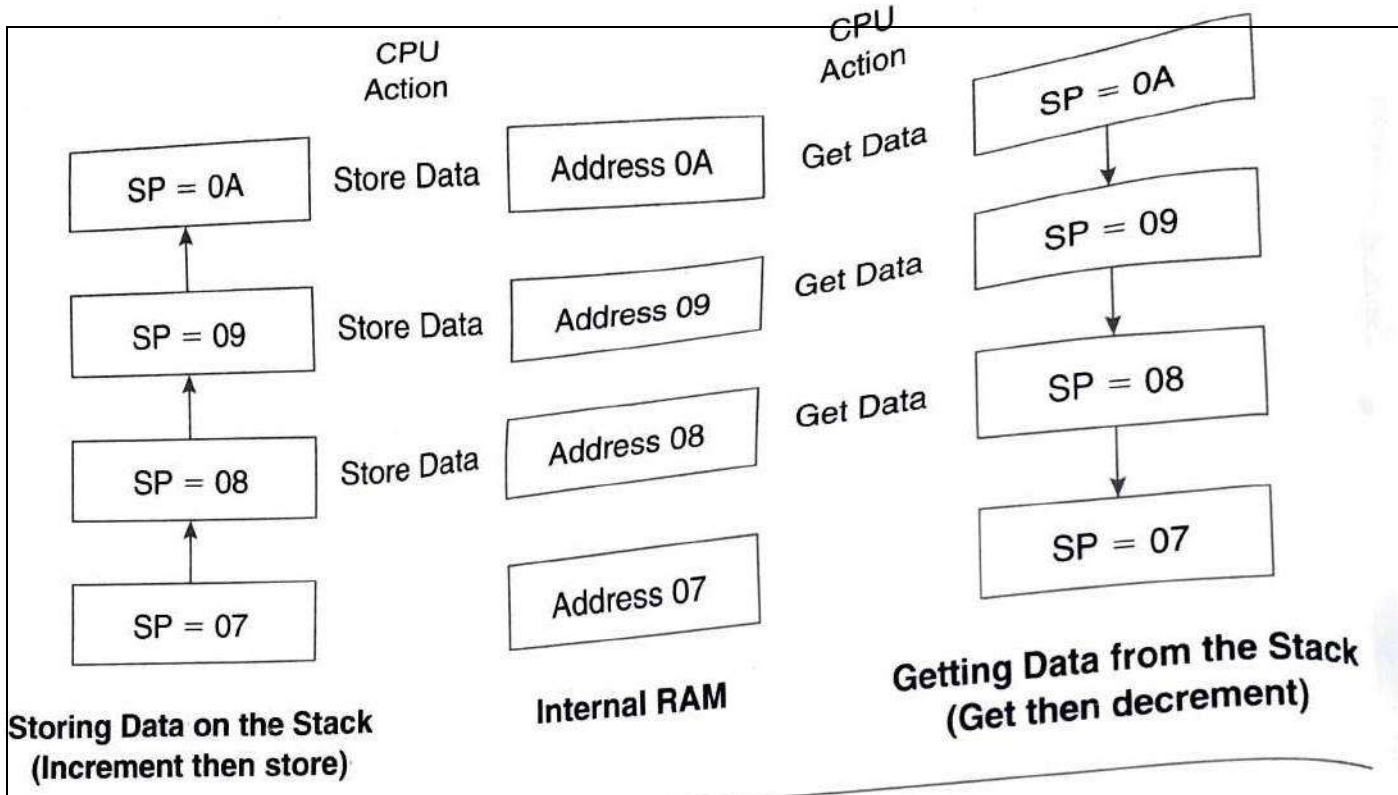


FIGURE 3.6 ◆ Stack Operation

In Figure 3.1b, the SFR addresses are shown in the upper right corner of each block. The SFR names and equivalent internal RAM addresses are given in Table 3.1. Note that the PC is not part of the SFR and has no internal RAM address. See also Appendix F.

SFRs are named in certain opcodes by their functional names, such as A or TH0, and are referenced by other opcodes by their addresses, such as 0E0h or 8Ch. Note that *any* address used in the program *must* start with a number; thus address E0h for the A SFR begins with 0. Failure to use this number convention will result in an assembler error when the program is assembled.

Internal ROM

The 8051 is organized so that data memory and program code memory can be in two entirely different physical memory entities. *Each* has the same address ranges.

The structure of the internal RAM has been discussed previously. A corresponding block of internal program code, contained in an internal ROM, occupies code address space 0000h to 0FFFh. The PC is ordinarily used to address program code bytes from addresses 0000h to FFFFh. Program addresses higher than 0FFFh, which exceed the internal ROM capacity, will cause the 8051 to automatically fetch code bytes from external program memory. Code bytes can

TABLE 3.1

Name	Function	Internal RAM Address (HEX)
A	Accumulator	0E0
B	Arithmetic	0F0
DPH	Addressing external memory	83
DPL	Addressing external memory	82
IE	Interrupt enable control	0A8
IP	Interrupt priority	0B8
P0	Input/output port latch	80
P1	Input/output port latch	90
P2	Input/output port latch	A0
P3	Input/output port latch	0B0
PCON	Power control	87
PSW	Program status word	0D0
SCON	Serial port control	98
SBUF	Serial port data buffer	99
SP	Stack pointer	81
TMOD	Timer/counter mode control	89
TCON	Timer/counter control	88
TL0	Timer 0 low byte	8A
TH0	Timer 0 high byte	8C
TL1	Timer 1 low byte	8B
TH1	Timer 1 high byte	8D

also be fetched exclusively from an external memory, addresses 0000h to FFFFh, by connecting the external access pin (EA pin 31 on the DIP) to ground. The PC does not care where the code is; the circuit designer decides whether the code is found totally in internal ROM, totally in external ROM, or in a combination of internal and external ROM.

1. Internal RAM
2. Internal special-function registers
3. External RAM
4. Internal and external ROM

Finally, the following five types of opcodes are used to move data:

1. MOV
2. MOVX
3. MOVC
4. PUSH and POP
5. XCH

◆ COMMENT ◆

All of the following opcode examples may be converted into operating programs that may be assembled and debugged by adding these code lines to the code found in each example:

```
        org 0000h       ; start code at 0000h
loop:                   ; a label to jump to (Chapter 8)
   (insert code here)
        sjmp loop       ; jump back to beginning
        end             ; end the program
```

5.1 Addressing Modes

The way the data sources or destination addresses are specified in the mnemonic that moves that data determines the addressing mode. Figure 5.1 diagrams the four addressing modes: immediate, register, direct, and indirect.

Immediate Addressing Mode

The simplest way to get data to a destination is to make the source of the data part of the opcode. The data source is then immediately available as part of the instruction itself.

When the 8051 executes an immediate data move, the program counter is automatically incremented to point to the byte(s) following the opcode byte in the program memory. Whatever data is found there is copied to the destination address.

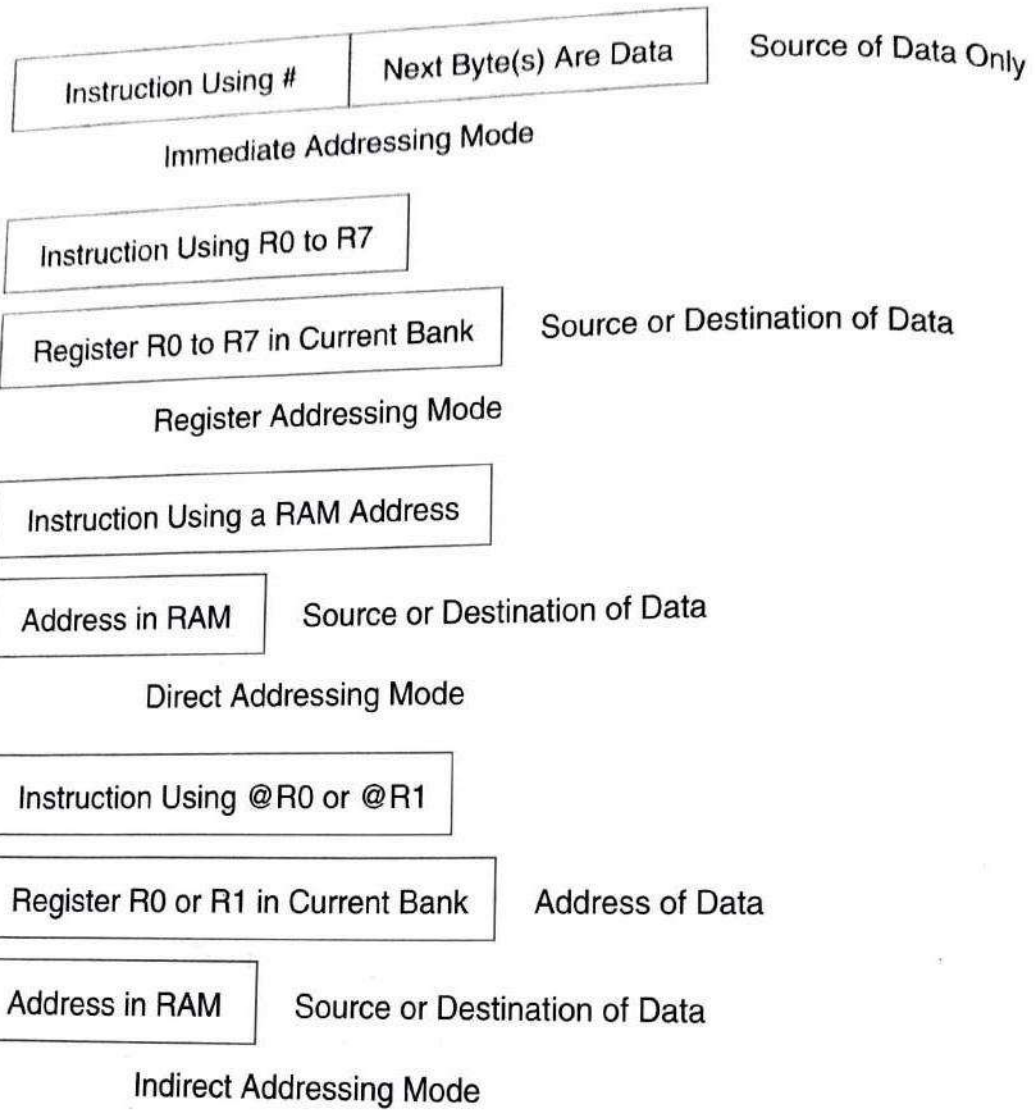


FIGURE 5.1 ◆ Addressing Modes

The mnemonic for immediate data is the pound sign (#). Occasionally, in the rush to meet a deadline, we might forget to use the # for immediate data. The resulting opcode is often a legal command that is assembled with no objections by the assembler. This omission guarantees that the rush will continue.

Three mnemonics can copy immediate numbers from the opcode into registers R0–R7 (of the currently selected register bank), A, and DPTR:

Mnemonic	Operation
MOV Rr,#n	Copy the 8-bit number n into register Rr (of the current register bank)
MOV A,#n	Copy the 8-bit number n into the Accumulator register
MOV DPTR,#nn	Copy the 16-bit number nn into the DPTR register

Immediate addressing modes, using some of the registers from R0 to R7 in the currently selected register bank, register A, and register DPTR, are shown in the following list:

Mnemonic	Operation
MOV R0,#00h	Put the immediate 8-bit number 00h in register R0
MOV R1,#01h	Put the immediate 8-bit number 01h in register R1
MOV R4,#04h	Put the immediate 8-bit number 04h in register R4
MOV R7,#07h	Put the immediate 8-bit number 07 in register R7
MOV A,#0AAh	Put the immediate 8-bit number AAh in register A
MOV DPTR,#1234h	Put the immediate 16-bit number 1234h in register DPTR

Register Addressing Mode

Certain register names may be used as part of the opcode mnemonic as sources or destinations of data. Registers A, DPTR, and R0 to R7 may be named as part of the opcode mnemonic. Other registers in the 8051 may be addressed using the direct addressing mode. Some assemblers can equate many of the direct addresses to the register name (as is the case with the assembler discussed in this book) so that register names may be used in lieu of register addresses. Remember that the registers used in the opcode as R0 to R7 are the ones that are *currently* chosen by the bank-select bits, RS0 and RS1 in the PSW.

Register-to-register moves are as follows:

Mnemonic	Operation
MOV A,Rr	Copy data from register Rr to register A
MOV Rr,A	Copy data from register A to register Rr

A data MOV does not alter the contents of the data source address. A *copy* of the data is made from the source and moved to the destination address. The contents of the destination address are replaced by the source address contents. The following list shows examples of MOV opcodes with immediate and register addressing modes:

Mnemonic	Operation
MOV A,#0F1h	Move the immediate data byte F1h to the A register
MOV A,R0	Copy the data in register R0 to register A
MOV DPTR,#0ABCDh	Move the immediate data bytes ABCDh to the DPTR
MOV R5,A	Copy the data in register A to register R5
MOV R3,#1Ch	Move the immediate data byte 1Ch to register R3

◆ CAUTION ◆

- It is impossible to have immediate data as a destination.
- All numbers *must* start with a decimal number (0 – 9), or the assembler assumes the number is a *label*.
- Register-to-register moves using the register addressing mode occur between registers A and R0 to R7.

Direct Addressing Mode

All 128 bytes of internal RAM and the SFRs may be addressed directly using the single-byte address assigned to each RAM location and each special-function register. See Appendix F for an overall SFR/memory map.

Internal RAM uses addresses from 00h to 7Fh to address each byte. The SFR addresses exist from 80h to FFh at the locations shown in Table 5.1.

◆ CAUTION ◆

- Note that there are "gaps" in the addresses of the SFRs; the addresses are not in order.
- Note the use of a leading 0 for all numbers that begin with an alphabetic (alpha) character.

RAM addresses 00 to 1Fh are *also* the locations assigned to the four banks of eight working registers, R0 to R7. This assignment means that R2 of register

TABLE 5.1

SFR	Address (hex)
A	0E0
B	0F0
DPL	82
DPH	83
IE	0A8
IP	0B8
P0	80
P1	90
P2	0A0
P3	0B0
PCON	87
PSW	0D0
SBUF	99
SCON	98
SP	81
TCON	88
TMOD	89
TH0	8C
TL0	8A
TH1	8D
TL1	8B

TABLE 5.2

Bank	Register	Address (hex)	Bank	Register	Address (hex)
0	R0	00	2	R0	10
0	R1	01	2	R1	11
0	R2	02	2	R2	12
0	R3	03	2	R3	13
0	R4	04	2	R4	14
0	R5	05	2	R5	15
0	R6	06	2	R6	16
0	R7	07	2	R7	17
1	R0	08	3	R0	18
1	R1	09	3	R1	19
1	R2	0A	3	R2	1A
1	R3	0B	3	R3	1B
1	R4	0C	3	R4	1C
1	R5	0D	3	R5	1D
1	R6	0E	3	R6	1E
1	R7	0F	3	R7	1F

bank 0 can be addressed in the register mode as R2 or in the direct mode as 02h. The direct addresses of the working registers are shown in Table 5.2.

Only one bank of working registers is *active* at any given time. The PSW special-function register holds the bank-select bits, RS0 and RS1, which determine which register bank is in use.

When the 8051 is reset, RS0 and RS1 are set to 00b to select the working registers in bank 0, located from 00h to 07h in internal RAM. Reset also sets SP to 07h, and the stack will grow *up* as it is used. This growing stack will overwrite the register banks above bank 0. Be *sure* to set the SP to a number above those of any working registers the program may use.

The programmer may choose any other bank by setting RS0 and RS1 as desired; this bank change is often done to "save" one bank and choose another when servicing an interrupt or using a subroutine.

The programmer may elect to use the *absolute* numeric address number for an SFR or may use a *symbol* (name) for the SFR. For example, the following instructions both move a *constant* number into port 1:

```
mov 90h,#0a5h
mov p1,#0a5h
```

The A51 assembler, supplied with this book, "looks up" the actual address of an SFR when the programmer uses an SFR symbol. Please refer to the end of Appendix B for a list of SFR symbols.

We shall use both methods of specifying SFRs in this book to emphasize the fact that SFRs are internal RAM addresses.

The moves made possible using direct, immediate, and register addressing modes are as follows:

Mnemonic	Operation
MOV A,add	Copy data from direct address add to register A
MOV add,A	Copy data from register A to direct address add
MOV Rr,add	Copy data from direct address add to register Rr
MOV add,Rr	Copy data from register Rr to direct address add
MOV add,#n	Copy immediate data byte n to direct address add
MOV add1,add2	Copy data from direct address add2 to direct address add1

The following list shows examples of MOV opcodes using direct, immediate, and register addressing modes:

Mnemonic	Operation
MOV A,80h	Copy data from the port 0 pins to register A
MOV 80h,A	Copy data from register A to the port 0 latch
MOV 3Ah,#3Ah	Copy immediate data byte 3Ah to RAM location 3Ah
MOV R0,12h	Copy data from RAM location 12h to register R0
MOV 8Ch,R7	Copy data from register R7 to timer 0 high byte
MOV 5Ch,A	Copy data from register A to RAM location 5Ch
MOV 0A8h,77h	Copy data from RAM location 77h to IE register

◆ CAUTION ◆

- MOV instructions that refer to direct addresses above 7Fh that are not SFRs will result in errors. The SFRs are physically on the chip; all other addresses above 7Fh do not physically exist.
- Moving data to a port changes the port *latch;* moving data from a port gets data from the port *pins.*
- Moving data from a direct address to itself is not predictable and could lead to errors.

Indirect Addressing Mode

For all the addressing modes covered to this point, the source or destination of the data is an absolute number or a name. Inspection of the opcode reveals exactly what the addresses are of the destination and source. For example, the opcode MOV A,R7 says that the A register will get a copy of whatever data is in register R7; MOV 33h,#32h moves the hex number 32 to hex RAM address 33.

The indirect addressing mode uses a register to *hold* the actual address that will finally be used in the data move; the register itself is *not* the address, but rather the number *in* the register. Indirect addressing for MOV opcodes uses register R0 or R1, often called a *data pointer,* to hold the address of one of the data locations in RAM from address 00h to 7Fh. The number that is in the pointing register (Rp) cannot be known unless the history of the register is known. The mnemonic symbol used for indirect addressing is the "at" sign, which is printed as @.

The moves made possible using immediate, direct, register, and indirect addressing modes are as follows:

Mnemonic	Operation
MOV @Rp,#n	Copy the immediate byte n to the address in Rp
MOV @Rp,add	Copy the contents of add to the address in Rp
MOV @Rp,A	Copy the data in A to the address in Rp
MOV add,@Rp	Copy the contents of the address in Rp to add
MOV A,@Rp	Copy the contents of the address in Rp to A

The following list shows examples of MOV opcodes, using immediate, register, direct, and indirect modes

Mnemonic	Operation
MOV A,@R0	Copy the contents of the address in R0 to the A register
MOV @R1,#35h	Copy the number 35h to the address in R1
MOV add,@R0	Copy the contents of the address in R0 to add
MOV @R1,A	Copy the contents of A to the address in R1
MOV @R0,80h	Copy the contents of the port 0 pins to the address in R0

◆ CAUTION ◆

- The number in register Rp must be a RAM address.
- *Only* registers R0 or R1 may be used for indirect addressing.

5.2 External Data Moves

As discussed in Chapter 3, it is possible to expand RAM and ROM memory space by adding external memory chips to the 8051 microcontroller. The external memory can be as large as 64K for each of the RAM and ROM memory areas. Opcodes that access this external memory *always* use indirect addressing to specify the external memory.

Figure 5.2 shows that registers R0, R1, and the aptly named DPTR can be used to hold the address of the data byte in external RAM. R0 and R1 are lim-

Mnemonic	Operation
MOV A,#3Ah	A = 3Ah
DEC A	A = 39h
MOV R0,#15h	R0 = 15h
MOV 15h,#12h	Internal RAM address 15h = 12h
INC @R0	Internal RAM address 15h = 13h
DEC 15h	Internal RAM address 15h = 12h
INC R0	R0 = 16h
MOV 16h,A	Internal RAM address 16h = 39h
INC @R0	Internal RAM address 16h = 3Ah
MOV DPTR,#12FFh	DPTR = 12FFh
INC DPTR	DPTR = 1300h
DEC 83h	DPTR = 1200h (SFR 83h is the DPH byte)

◆ CAUTION ◆

- Remember: *No* math flags are affected.
- All 8-bit address contents overflow from FFh to 00h.
- DPTR is 16 bits; DPTR overflows from FFFFh to 0000h.
- The 8-bit address contents underflow from 00h to FFh.
- There is no DEC DPTR to match the INC DPTR.

7.3 Addition

All addition is done with the A register as the destination of the result. All addressing modes may be used for the source: an immediate number, a register, a direct address, and an indirect address. Some instructions include the Carry flag as an additional source of a single bit that is included in the operation at the *least* significant bit position.

The following list shows the addition mnemonics:

Mnemonic	Operation
ADD A,#n	Add A and the immediate number n; put the sum in A
ADD A,Rr	Add A and register Rr; put the sum in A
ADD A,add	Add A and the address contents; put the sum in A
ADD A,@Rp	Add A and the contents of the address in Rp; put the sum in A

Note that the C flag is set to 1 if there is a carry out of bit position 7; it is cleared to 0 otherwise. The AC flag is set to 1 if there is a carry out of bit position 3; it is cleared otherwise. The OV flag is set to 1 if there is a carry out of bit

position 7, but not bit position 6 or if there is a carry out of bit position 6 but not bit position 7, which may be expressed as the logical operation

OV = C7 XOR C6

Unsigned and Signed Addition

The programmer may decide that the numbers used in the program are to be unsigned numbers—that is, numbers that are 8-bit positive binary numbers ranging from 00h to FFh. Alternatively, the programmer may need to use both positive and negative signed numbers.

Signed numbers use bit 7 as a *sign* bit in the most significant byte (MSB) of the *group* of bytes *chosen* by the programmer to represent the largest number to be needed by the program. Bits 0 to 6 of the MSB, and any other bytes, express the magnitude of the number. Signed numbers use a 1 in bit position 7 of the MSB as a negative sign and a 0 as a positive sign. Further, all negative numbers are not in true form, but are in two's complement form. When doing signed arithmetic, the programmer must *know* how large the largest number is to be—that is, how many bytes are needed for each number.

In signed form, a single byte number may range in size from 10000000b, which is −128d, to 01111111b, which is +127d. The number 00000000b is 000d and has a positive sign, so there are 128d negative numbers and 128d positive numbers. The C and OV flags have been included in the 8051 to enable the programmer to use either numbering scheme.

Adding or subtracting unsigned numbers may generate a Carry flag when the sum exceeds FFh or a Borrow flag when the minuend is less than the subtrahend. The OV flag is not used for unsigned addition and subtraction. Adding or subtracting signed numbers can lead to carries and borrows in a similar manner, and to overflow conditions as a result of the actions of the sign bits.

Unsigned Addition

Unsigned numbers make use of the Carry flag to detect when the result of an ADD operation is a number larger than FFh. If the carry is set to 1 after an ADD, then the carry can be added to a higher order byte so that the sum is not lost. For instance:

```
 95d =    01011111b          =   5Fh
189d =    10111101b          =   BDh
284d    1)00011100b = 284d       1)1Ch
```

where) indicates the state of the C flag. The C flag is set to 1 to account for the carry out from the sum. The program could add the Carry flag to another byte that forms the second byte of a larger number.

Signed Addition

Signed numbers may be added two ways: addition of like signed numbers and addition of unlike signed numbers. If unlike signed numbers are added, then it is not possible for the result to be larger than −128d or +127d, and the sign of the result will always be correct. For example:

−001d	=	11111111b	=	FFh
+027d	=	00011011b	=	1Bh
+026d		1)00011010b	= +026d	1)1Ah

Here, there is a carry from bit 7 so the Carry flag is 1. There is also a carry from bit 6, and the OV flag is 0. For this condition, no action need be taken by the program to correct the sum.

If positive numbers are added, there is the possibility that the sum will exceed +127d, as demonstrated in the following example:

+100d	=	01100100b	=	64h
+050d	=	00110010b	=	32h
+150d		0)10010110b	= −106d	0)96h

Ignoring the sign of the result, the magnitude is seen to be +22d, which would be correct if we had some way of accounting for the +128d, which, unfortunately, is larger than a single byte can hold. There is no carry from bit 7 and the Carry flag is 0; there is a carry from bit 6 so the OV flag is 1.

An example of adding two positive numbers that do not exceed the positive limit is this:

+045d	=	00101101b	=	2Dh
+075d	=	01001011b	=	4Bh
+120d		0)01111000b	= 120d	0)78h

Note that there are no carries from bits 6 or 7 of the sum; the Carry and OV flags are both 0.

The result of adding two negative numbers together for a sum that does not exceed the negative limit is shown in this example:

−030d	=	11100010b	=	E2h
−050d	=	11001110b	=	CEh
−080d		1)10110000b	= −080d	1)BCh

Here, there is a carry from bit 7 and the Carry flag is 1; there is a carry from bit 6 and the OV flag is 0. These are the same flags as the case for adding unlike numbers; no corrections are needed for the sum.

When adding two negative numbers whose sum does exceed −128d, we have:

−070d	=	10111010b	=	BAh
−070d	=	10111010b	=	BAh
−140d		1)01110100b = +116d		1)74h

Or, the magnitude can be interpreted as −12d, which is the remainder after a carry out of −128d. In this example, there is a carry from bit position 7, and no carry from bit position 6, so the Carry and the OV flags are set to 1. The magnitude of the sum is correct; the sign bit must be changed to a 1.

From these examples the programming actions needed for the C and OV flags are as follows:

Flags		Action
C	OV	
0	0	None
0	1	Complement the sign
1	0	None
1	1	Complement the sign

A general rule is that *if the OV flag is set, then complement the sign.* The OV flag also signals that the sum exceeds the largest positive or negative numbers thought to be needed in the program.

Multiple-Byte Signed Arithmetic

The nature of multiple-byte arithmetic for signed and unsigned numbers is distinctly different from single-byte arithmetic. Using more than one byte in unsigned arithmetic means that carries or borrows are propagated from low-order to high-order bytes by the simple technique of adding the carry to the next highest byte for addition and subtracting the borrow from the next highest byte for subtraction.

Signed numbers appear to behave like unsigned numbers until the last byte is reached. For a signed number, the seventh bit of the highest byte is the sign; if the sign is negative, then the *entire* number is in two's complement form.

For example, using a 2-byte signed number, we have the following examples:

+32767d	01111111 11111111b	7FFFh
+00000d	00000000 00000000b	0000h
−00001d	11111111 11111111b	FFFFh
−32768d	10000000 00000000b	8000h

Note that the lowest byte of the numbers 00000d and −32768d are exactly alike, as are the lowest bytes for +32767d and −00001d.

For multi-byte signed number arithmetic, then, the lower bytes are treated as unsigned numbers. All checks for overflow are done only for the highest order byte that contains the sign. An overflow at the highest order byte is not used

ally recoverable. The programmer has made a *mistake* and probably has made no provisions for a number larger than planned. Some error acknowledgment procedure, or user notification, should be included in the program if this type of mistake is a possibility.

The preceding examples show the need to add the Carry flag to higher order bytes in signed and unsigned addition operations. Opcodes that accomplish this task are similar to the ADD mnemonics: A C is appended to show that the carry bit is added to the sum in bit position 0.

The following list shows the add with carry mnemonics:

Mnemonic	Operation
ADDC A,#n	Add the contents of A, the immediate number n, and the C flag; put the sum in A
ADDC A,add	Add the contents of A, the direct address contents, and the C flag; put the sum in A
ADDC A,Rr	Add the contents of A, register Rr, and the C flag; put the sum in A
ADDC A,@Rp	Add the contents of A, the contents of the indirect address in Rp, and the C flag; put the sum in A

Note that the C, AC, and OV flags behave exactly as they do for the ADD commands.

The following list shows examples of ADD and ADDC multiple-byte signed arithmetic operations:

Mnemonic	Operation
MOV A,#1Ch	A = 1Ch
MOV R5,#0A1h	R5 = A1h
ADD A,R5	A = BDh; C = 0, OV = 0
ADD A,R5	A = 5Eh; C = 1, OV = 1
ADDC A,#10h	A = 6Fh; C = 0, OV = 0
ADDC A,#10h	A = 7Fh; C = 0, OV = 0

◆ CAUTION ◆

- ADDC is normally used to add a carry after the LSBY addition in a multi-byte process. ADD is normally used for the LSBY addition.

7.4 Subtraction

Subtraction can be done by taking the two's complement of the number to be subtracted, the subtrahend, and adding it to another number, the minuend. The 8051, however, has commands to perform direct subtraction of two signed or unsigned numbers. Register A is the destination address for subtraction. All four addressing modes may be used for source addresses. The commands treat the Carry flag as a borrow and always subtract the Carry flag as part of the operation.

The following list shows the subtract mnemonics:

Mnemonic	Operation
SUBB A,#n	Subtract immediate number n and the C flag from A; put the result in A
SUBB A,add	Subtract the contents of add and the C flag from A; put the result in A
SUBB A,Rr	Subtract Rr and the C flag from A; put the result in A
SUBB A,@Rp	Subtract the contents of the address in Rp and the C flag from A; put the result in A

Note that the C flag is set if a borrow is needed into bit 7 and reset otherwise. The AC flag is set if a borrow is needed into bit 3 and reset otherwise. The OV flag is set if there is a borrow into bit 7 and not bit 6 or if there is a borrow into bit 6 and not bit 7. As in the case for addition, the OV flag is the XOR of the borrows into bit positions 7 and 6.

Unsigned and Signed Subtraction

Again, depending on what is needed, the programmer may choose to use bytes as signed or unsigned numbers. The Carry flag is now thought of as a Borrow flag to account for situations when a larger number is subtracted from a smaller number. The OV flag indicates results that must be adjusted whenever two numbers of unlike signs are subtracted and the result exceeds the planned signed magnitudes.

Unsigned Subtraction

Because the C flag is always subtracted from A along with the source byte, it must be set to 0 if the programmer does not want the flag included in the subtraction. If a multi-byte subtraction is done, the C flag is cleared for the first byte and then included in subsequent higher byte operations.

The result will be in true form, with no borrow if the source number is smaller than A, or in two's complement form, with a borrow if the source is larger than A. These are *not* signed numbers, as all 8 bits are used for the magnitude. The range of numbers is from positive 255d (C = 0, A = FFh) to negative 255d (C = 1, A = 01h).

The following example demonstrates subtraction of a larger number from a smaller number:

```
        015d =   00001111b          =   0Fh
SUBB    100d =   01100100b          =   64h
       -085d     1)10101011b = 171d  = 1)ABh
```

The C flag is set to 1, and the OV flag is set to 0. The two's complement of the result is 085d.

The reverse of the example yields the following result:

```
      100d =    01100100b          =   64h
SUBB  015d =    00001111b          =   0Fh
      085d =  0)01010101b = 085d       0)55h
```

The C flag is set to 0, and the OV flag is set to 0. The magnitude of the result is in true form.

Signed Subtraction

As is the case for addition, two combinations of unsigned numbers are possible when subtracting: subtracting numbers of like and unlike signs. When numbers of like sign are subtracted, it is impossible for the result to exceed the positive or negative magnitude limits of +127d or −128d, so the magnitude and sign of the result do not need to be adjusted, as shown in the following example:

```
      +100d =    01100100b (Carry flag = 0 before SUBB)  =   64h
SUBB  +126d =    01111110b                               =   7Eh
      −026d    1)11100110b = −026d                           1)E6h
```

There is a borrow into bit positions 7 and 6; the Carry flag is set to 1, and the OV flag is cleared.

The following example demonstrates using two negative numbers:

```
      −061d =    11000011b (Carry flag = 0 before SUBB)  =   C3h
SUBB  −116d =    10001100b                               =   8Ch
      +055d    0)00110111b = +55d                            0)37h
```

There are no borrows into bit positions 6 or 7, so the OV and Carry flags are cleared to 0.

An overflow is possible when subtracting numbers of opposite sign because the situation becomes one of adding numbers of like signs, as can be demonstrated in the following example:

```
      −099d =    10011101b (Carry flag = 0 before SUBB)  =   9Dh
SUBB  +100d =    01100100b                               =   64h
      −199d    0)00111001b = +057d                           0)39h
```

Here, there is a borrow into bit position 6 but not into bit position 7; the OV flag is set to 1, and the Carry flag is cleared to 0. Because the OV flag is set to 1, the result must be adjusted. In this case, the magnitude can be interpreted as the two's complement of 71d, the remainder after a carry out of 128d from 199d. The magnitude is correct, and the sign needs to be corrected to a 1.

The following example shows a positive overflow:

```
        +087d =      01010111b (Carry flag = 0 before SUBB) =   57h
SUBB    −052d =      11001100b                              =   CCh
        +139d        1)10001011b = −117d                        1)8Bh
```

There is a borrow from bit position 7, and no borrow from bit position 6; the OV flag and the Carry flag are both set to 1. Again the answer must be adjusted because the OV flag is set to 1. The magnitude can be interpreted as a +011d, the remainder from a carry out of 128d. The sign must be changed to a binary 0 and the OV condition dealt with.

The general rule is that *if the OV flag is set to 1, then complement the sign bit.* The OV flag also signals that the result is greater than −128d or +127d.

Again, it must be emphasized: When an overflow occurs in a program, an error has been made in the estimation of the largest number needed to successfully operate the program. Theoretically, the program could resize every number used, but this extreme procedure would tend to hinder the performance of the microcontroller.

Note that for all the examples in this section, it is *assumed* that the Carry flag = 0 before the SUBB. The Carry flag must be 0 before any SUBB operation that depends on C = 0 is done.

The following list shows examples of SUBB multiple-byte signed arithmetic operations:

Mnemonic	Operation
MOV 0D0h,#00h	Carry flag = 0
MOV A,#3Ah	A = 3Ah
MOV 45h,#13h	Address 45h = 13h
SUBB A,45h	A = 27h; C = 0, OV = 0
SUBB A,45h	A = 14h; C = 0, OV = 0
SUBB A,#80h	A = 94h; C = 1, OV = 1
SUBB A,#22h	A = 71h; C = 0, OV = 0
SUBB A,#0FFh	A = 72h; C = 1, OV = 0

◆ CAUTION ◆

- Remember to set the Carry flag to 0 if it is not to be included as part of the subtraction operation.

7.5 Multiplication and Division

The 8051 has the capability to perform 8-bit integer multiplication and division using the A and B registers. Register B is used solely for these operations and has no other use except as a location in the SFR space of RAM that could

be used to hold data. The A register holds 1 byte of data before a multiply or divide operation, and 1 of the result bytes after a multiply or divide operation.

Multiplication and division treat the numbers in registers A and B as unsigned. The programmer must devise ways to handle signed numbers.

Multiplication

Multiplication operations use registers A and B as both source and destination addresses for the operation. The unsigned number in register A is multiplied by the unsigned number in register B, as follows:

Mnemonic	Operation
MUL AB	Multiply A by B; put the low-order byte of the product in A, put the high-order byte in B

The OV flag will be set if A × B > FFh. Setting the OV flag does *not* mean that an error has occurred. Rather, it signals that the number is larger than 8 bits, and the programmer needs to inspect register B for the high-order byte of the multiplication operation. The Carry flag is always cleared to 0.

The largest possible product is FE01h when both A and B contain FFh. Register A contains 01h and register B contains FEh after multiplication of FFh by FFh. The OV flag is set to 1 to signal that register B contains the high-order byte of the product; the Carry flag is 0.

The following list gives examples of MUL multiple-byte arithmetic operations:

Mnemonic	Operation
MOV A,#7Bh	A = 7Bh
MOV 0F0h,#02h	B = 02h
MUL AB	A = F6h and B = 00h; OV Flag = 0
MOV B,#0FEh	B = FEh
MUL AB	A = 14h and B = F4h; OV Flag = 1

◆ CAUTION ◆

- Note there is no comma between A and B in the MUL mnemonic.

Division

Division operations use registers A and B as both source and destination addresses for the operation. The unsigned number in register A is divided by the unsigned number in register B, as follows:

Mnemonic	Operation
DIV AB	Divide A by B; put the integer part of quotient in register A and the integer part of the remainder in B

The OV flag is cleared to 0 unless B holds 00h before the DIV. Then the OV flag is set to 1 to show division by 0. The contents of A and B, when division by 0 is attempted, are undefined. The Carry flag is always reset.

Division always results in integer quotients and remainders, as shown in the following example:

$$\frac{A1=213d}{B1=017d} = 12 \text{ (quotient) and } 9 \text{ (remainder)}$$
$$[213d = (12 \times 17) + 9]$$

When done in hex:

$$\frac{A=0D5h}{B=011h} = C \text{ (quotient) and } 9 \text{ (remainder)}$$

The following list gives examples of DIV multiple-byte arithmetic operations:

Mnemonic	Operation
MOV A,#0FFh	A = FFh (255d)
MOV 0F0h,#2Ch	B = 2C (44d)
DIV AB	A = 05h and B = 23h [255d = (5 × 44) + 35]
DIV AB	A = 00h and B = 05h [05d = (0 × 35) + 5]
DIV AB	A = 00h and B = 00h [00d = (0 × 5) + 0]
DIV AB	A = ?? and B = ??; OV flag is set to 1

◆ CAUTION ◆

- The original contents of A and B are lost.
- Note there is no comma between A and B in the DIV mnemonic.

6 Decimal Arithmetic

Most 8051 applications involve adding intelligence to machines where the hexadecimal numbering system works naturally. There are instances, however, when the application involves interacting with humans, who insist on using the decimal number system. In such cases, it may be more convenient for the programmer to use the decimal number system to represent all numbers in the program.

Four bits are required to represent the decimal numbers from 0 to 9 (0000 to 1001) and the numbers are often called *binary coded decimal* (BCD) numbers. Two of these BCD numbers can then be packed into a single byte of data.

The 8051 does all arithmetic operations in pure binary. When BCD numbers are being used the result will often be a non-BCD number, as shown in the following example:

49BCD	= 01001001b		= 49h
+38BCD	= 00111000b		= 38h
87BCD	10000001b = 81BCD		81h

Jumps and calls may also be generically referred to as "branches," which emphasizes that two divergent paths are made possible by this type of instruction.

8.1 The Jump and Call Program Range

A jump or call instruction can replace the contents of the program counter with a new program address number that causes program execution to begin at the code located at the new address. The difference, in bytes, of this new address from the address in the program where the jump or call is located is called the *range* of the jump or call. For example, if a jump instruction is located at program address 0100h, and the jump causes the program counter to become 0120h, then the range of the jump is 20h bytes.

Jump or call instructions may have one of three ranges: a *relative* range of +127d, −128d bytes from the instruction *following* the jump or call instruction; an *absolute* range on the same 2K byte page as the instruction *following* the jump or call; or a *long* range of any address from 0000h to FFFFh, anywhere in program memory. Figure 8.1 shows the relative range of all the jump instructions.

Relative Range

Jumps that replace the program counter contents with a new address that is greater than the address of the instruction *following* the jump by 127d or less than the address of the instruction following the jump by 128d are called *relative* jumps. They are so named because the address that is placed in the program counter is relative to the address where the jump occurs. If the absolute address of the jump instruction changes, then the jump address changes also but remains the same distance away from the jump instruction. The address following the jump is used to calculate the relative jump because of the action of the PC. The PC is incremented to point to the *next* instruction *before* the current instruction is executed. Thus, the PC is set to the following address before the jump instruction is executed, or in the vernacular: "before the jump is taken."

Relative jumping has two advantages. First, only 1 byte of data need be specified, either in positive format for jumps ahead in the program or in two's complement negative format for jumps behind. The jump address displacement byte can then be added to the PC to get the absolute address. Specifying only 1 byte saves program bytes and speeds up program execution. Second, the program that is written using relative jumps can be located anywhere in the program address space without re-assembling the code to generate absolute addresses.

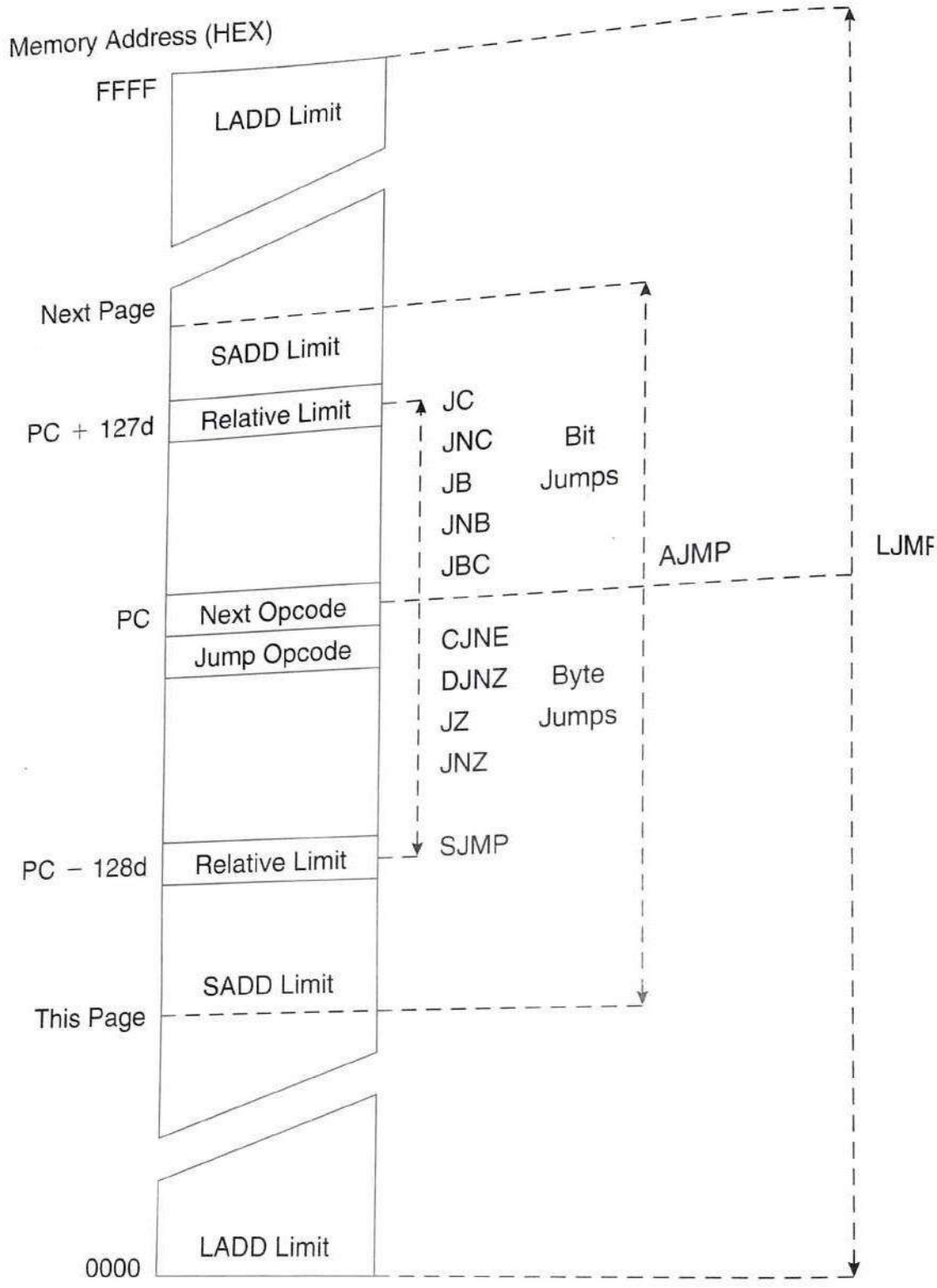


FIGURE 8.1 ◆ Jump Instruction Ranges

The disadvantage of using relative addressing is the requirement that all addresses jumped be within a range of +127d, −128d bytes (an 8-bit signed number range) of the jump instruction. This range is not a serious problem. Most jumps form program loops over short code ranges that are within the relative address capability. Jumps are the only branch instructions that can use the relative range.

If jumps beyond the relative range are needed, then a relative jump can be done to another relative jump until the desired address is reached. This need is better handled, however, by the jumps that are covered in the next sections.

Short Absolute Range

Absolute range makes use of the concept of dividing memory into logical divisions called *pages*. Program memory may be regarded as one continuous stretch of addresses from 0000h to FFFFh. Or, it may be divided into a series of pages of any convenient binary size, such as 256 bytes, 2K, 4K, and so on.

The 8051 program memory is arranged as 2K pages, giving a total of 32d (20h) pages. The hexadecimal address of each page is shown in Table 8.1.

Inspection of the page numbers shows that the upper 5 bits of the program counter hold the page *number*, and the lower 11 bits hold the *address* within each page. An absolute address is formed by taking the page number of the instruction *following* the branch and attaching the absolute page range address of 11 bits to it to form the 16-bit address.

Branches on page *boundaries* occur when the jump or call instruction finishes at X7FFh or XFFFh. The next instruction starts at X800h or X000h, which places the jump or call address on the same page as the *next* instruction after the jump or call. The page change presents no problem when branching ahead but could be troublesome if the branch is *backwards* in the program. The assembler will flag such problems as errors, so adjustments can be made by the programmer to use a different type of range.

Absolute range addressing has the same advantages as relative addressing; fewer bytes are needed and the code is relocatable as long as the relocated code remains on the following page. Absolute addressing has the advantage of

TABLE 8.1

Page	Address (hex)	Page	Address (hex)	Page	Address (hex)
00	0000 – 07FF	0B	5800 – 5FFF	16	B000 – B7FF
01	0800 – 0FFF	0C	6000 – 67FF	17	B800 – BFFF
02	1000 – 17FF	0D	6800 – 6FFF	18	C000 – C7FF
03	1800 – 1FFF	0E	7000 – 77FF	19	C800 – CFFF
04	2000 – 27FF	0F	7800 – 7FFF	1A	D000 – D7FF
05	2800 – 2FFF	10	8000 – 87FF	1B	D800 – DFFF
06	3000 – 37FF	11	8800 – 8FFF	1C	E000 – E7FF
07	3800 – 3FFF	12	9000 – 97FF	1D	E800 – EFFF
08	4000 – 47FF	13	9800 – 9FFF	1E	F000 – F7FF
09	4800 – 4FFF	14	A000 – A7FF	1F	F800 – FFFF
0A	5000 – 57FF	15	A800 – AFFF		

Short Absolute Range

Absolute range makes use of the concept of dividing memory into logical divisions called *pages*. Program memory may be regarded as one continuous stretch of addresses from 0000h to FFFFh. Or, it may be divided into a series of pages of any convenient binary size, such as 256 bytes, 2K, 4K, and so on.

The 8051 program memory is arranged as 2K pages, giving a total of 32d (20h) pages. The hexadecimal address of each page is shown in Table 8.1.

Inspection of the page numbers shows that the upper 5 bits of the program counter hold the page *number*, and the lower 11 bits hold the *address* within each page. An absolute address is formed by taking the page number of the instruction *following* the branch and attaching the absolute page range address of 11 bits to it to form the 16-bit address.

Branches on page *boundaries* occur when the jump or call instruction finishes at X7FFh or XFFFh. The next instruction starts at X800h or X000h, which places the jump or call address on the same page as the *next* instruction after the jump or call. The page change presents no problem when branching ahead but could be troublesome if the branch is *backwards* in the program. The assembler will flag such problems as errors, so adjustments can be made by the programmer to use a different type of range.

Absolute range addressing has the same advantages as relative addressing; fewer bytes are needed and the code is relocatable as long as the relocated code remains on the following page. Absolute addressing has the advantage of

TABLE 8.1

Page	Address (hex)	Page	Address (hex)	Page	Address (hex)
00	0000 – 07FF	0B	5800 – 5FFF	16	B000 – B7FF
01	0800 – 0FFF	0C	6000 – 67FF	17	B800 – BFFF
02	1000 – 17FF	0D	6800 – 6FFF	18	C000 – C7FF
03	1800 – 1FFF	0E	7000 – 77FF	19	C800 – CFFF
04	2000 – 27FF	0F	7800 – 7FFF	1A	D000 – D7FF
05	2800 – 2FFF	10	8000 – 87FF	1B	D800 – DFFF
06	3000 – 37FF	11	8800 – 8FFF	1C	E000 – E7FF
07	3800 – 3FFF	12	9000 – 97FF	1D	E800 – EFFF
08	4000 – 47FF	13	9800 – 9FFF	1E	F000 – F7FF
09	4800 – 4FFF	14	A000 – A7FF	1F	F800 – FFFF
0A	5000 – 57FF	15	A800 – AFFF		

addressing.

Long Absolute Range

Addresses that can access the entire program space from 0000h to FFFFh use long-range addressing. Long-range addresses require more bytes of code to specify and are relocatable only at the beginning of 64K pages. Since we are limited to a nominal ROM address range of 64K, the program must be re-assembled every time a long-range address changes and these branches are not generally relocatable.

Long-range addressing has the advantage of using the entire program address space available to the 8051. It is most likely to be used in large programs.

3.2 Jumps

The ability of a program to respond quickly to changes in conditions depends largely on the number and types of jump instructions available to the programmer. The 8051 has a rich set of jumps that can operate at the bit and byte levels. These jump opcodes are one reason the 8051 is such a powerful microcontroller.

Jumps operate by testing for conditions that are specified in the jump mnemonic. If the condition is *true*, then the jump is taken—that is, the program counter is altered to the address that is part of the jump instruction. If the condition is *false*, then the instruction immediately following the jump instruction is executed because the program counter is not altered. Keep in mind that the condition of *true* does *not* mean a binary 1 and that *false* does *not* mean binary 0. The *condition* specified by the mnemonic is either true or false.

Bit Jumps

Bit jumps all operate according to the status of the Carry flag in the PSW or the status of any bit-addressable location. All bit jumps are relative to the program counter.

Jump instructions that test for bit conditions are shown in the following list:

Mnemonic	Operation
JC radd	Jump relative if the Carry flag is set to 1
JNC radd	Jump relative if the Carry flag is reset to 0
JB b,radd	Jump relative if addressable bit is set to 1
JNB b,radd	Jump relative if addressable bit is reset to 0
JBC b,radd	Jump relative if addressable bit is set, and clear the addressable bit to 0

Note that no flags are affected unless the bit in JBC is a flag bit in the PSW. When the bit used in a JBC instruction is a port bit, the SFR latch for that port is read, tested, and altered.

The following program example makes use of bit jumps:

```
Address   Mnemonic          Comment
LOOP:     MOV A,#10h        ;A = 10h
          MOV R0,A          ;R0 = 10h
ADDA:     ADD A,R0          ;add R0 to A
          JNC ADDA          ;if the Carry flag is 0, then no carry is
                            ;true; jump to address ADDA; jump until A
                            ;is F0h; the C flag is set to
                            ;1 on the next ADD and no carry is
                            ;false; do the next instruction
          MOV A,#10h        ;A = 10h; do program again using JNB
ADDR:     ADD A,R0          ;add R0 to A (R0 already equals 10h)
          JNB 0D7h,ADDR     ;D7h is the bit address of the Carry flag
          JBC 0D7h,LOOP     ;the carry bit is 1; the jump to LOOP
                            ;is taken, and the Carry flag is cleared
                            ;to 0
END
```

◆ CAUTION ◆

- All jump addresses, such as ADDA and ADDR, must be within +127d, −128d of the instruction following the jump opcode.
- If the addressable bit is a flag bit and JBC is used, the flag bit will be cleared.
- *Do not* use any label names that are also the names of registers in the 8051. These are called *reserved* words and will cause great agitation in the assembler. See Appendix B for a listing of reserved names (also called *symbols*.)

Byte Jumps

Byte jumps—jump instructions that test bytes of data—behave as bit jumps. If the condition that is tested is *true,* the jump is taken; if the condition is *false,* the instruction after the jump is executed. All byte jumps are relative to the program counter.

The following list shows examples of byte jumps:

Mnemonic	Operation
CJNE A,add,radd	Compare the contents of the A register with the contents of the direct address; if they are *not* equal, then jump to the relative address; set the Carry flag to 1 if A is less than the contents of the direct address; otherwise, set the Carry flag to 0
CJNE A,#n,radd	Compare the contents of the A register with the immediate number n; if they are *not* equal, then jump to the relative address; set the Carry flag to 1 if A is less than the number; otherwise, set the Carry flag to 0
CJNE Rr,#n,radd	Compare the contents of register Rr with the immediate number n; if they are *not* equal, then jump to the relative address; set the Carry flag to 1 if Rn is less than the number; otherwise, set the Carry flag to 0
CJNE @Rp,#n,radd	Compare the contents of the address contained in register Rp to the number n; if they are *not* equal, then jump to the relative address; set the Carry flag to 1 if the contents of the address in Rp are less than the number; otherwise, set the Carry flag to 0
DJNZ Rr,radd	Decrement register Rr by 1 and jump to the relative address if the result is *not* 0; no flags are affected
DJNZ add,radd	Decrement the direct address by 1 and jump to the relative address if the result is *not* 0; no flags are affected unless the direct address is the PSW
JZ radd	Jump to the relative address if A is 0; the flags and the A register are not changed
JNZ radd	Jump to the relative address if A is *not* 0; the flags and the A register are not changed

Note that if the direct address used in a DJNZ is a port, the port SFR is decremented and tested for 0.

Unconditional Jumps

Unconditional jumps do not test any bit or byte to determine whether the jump should be taken. The jump is *always* taken. All jump ranges are found in this group of jumps, and these are the only jumps that can jump to any location in memory.

The following list shows examples of unconditional jumps:

Mnemonic	Operation
JMP @A+DPTR	Jump to the address formed by adding A to the DPTR; this is an unconditional jump and will always be done; the address can be anywhere in program memory; A, the DPTR, and the flags are unchanged
AJMP sadd	Jump to absolute short range address *sadd;* this is an unconditional jump and is always taken; no flags are affected
LJMP ladd	Jump to absolute long range address *ladd;* this is an unconditional jump and is always taken; no flags are affected
SJMP radd	Jump to relative address *radd;* this is an unconditional jump and is always taken; no flags are affected
NOP	Do nothing and go to the next instruction; NOP (no operation) is used to waste time in a software timing loop, or to leave room in a program for later additions; no flags are affected

The Binary Number System

A binary number system uses two decimal numerals, 0 and 1. A binary number is made up of a collection of binary numerals using a radix of 2. Binary digits are also called *bits,* which is a contraction of the words *Binary* and *digITS.*

In an electronic circuit, 1 and 0 might correspond to a switch that is off or on, or to a circuit that has a high voltage or a low voltage output. The assignment of binary 1 to off, for example, or binary 0 to a high voltage is arbitrary. A binary 1 may correspond to a switch that is off, or one that is on. Standards for the computer industry, however, have generally followed the practice that a circuit that is in a high-voltage state is a binary 1, and one that is in a low-voltage state is binary 0.

Conversions Between Decimal and Binary Numbers

The decimal equivalent of a binary number is formed by multiplying each bit in the binary number by the binary radix of 2 raised to the position's power. The result of each multiplication is expressed as a decimal number. The individual decimal numbers are added to obtain the decimal equivalent of the binary number. For instance, the binary number 111110100 can be converted to a decimal equivalent number as shown in Table 2.6.

The reverse process, that of converting a number with a decimal radix to a number with a binary radix, is done by repeated division of the decimal number by the binary radix of 2. The decimal number is repeatedly divided by 2, and the remainder becomes a bit of the equivalent binary number. Any decimal number divided by 2 must have a remainder of 0 if the decimal number is even, or 1 if the decimal number is odd.

TABLE 2.6

Digit	Position	2^{Position}	Decimal Equivalent
1	8	256	$1 \times 256 = 256$
1	7	128	$1 \times 128 = 128$
1	6	64	$1 \times 64 = 64$
1	5	32	$1 \times 32 = 32$
1	4	16	$1 \times 16 = 16$
0	3	8	$0 \times 8 = 0$
1	2	4	$1 \times 4 = 4$
0	1	2	$0 \times 2 = 0$
0	0	1	$0 \times 1 = 0$
			500

The remainder of the first division operation yields the *least significant bit* (LSB) of the binary number. The number left after the first division by 2 (the first quotient) is again divided by 2, yielding a second remainder and a second quotient. The second remainder is the next most significant bit of the binary equivalent number. The process of dividing the quotient by 2 and keeping the remainder as a bit of the binary equivalent number is continued until the *quotient* is zero. The final remainder is the *most significant bit* (MSB) of the binary equivalent number.

To demonstrate conversion from a decimal number to an equivalent binary number, we shall take the decimal 500 number from the last example and reconvert it to binary. See Table 2.7.

Hexadecimal Numbers

The *larger* the radix of a number system, the more *compact* is the expression for any number in the system. Because each digit is multiplied by a larger radix raised to the positional power of the digit, the value of the number grows rapidly as digits are added.

A radix 5 number is more compact than a binary number, and a decimal radix number is more compact than a radix 5 number. For example, the number of digits required to express decimal 57 in systems of radix 2, 3, and 5 are shown next:

111001 radix 2 = 57 decimal
2010 radix 3 = 57 decimal
212 radix 5 = 57 decimal

Radix 12 numbers, in turn, are even more compact than decimal numbers. But, a problem arises when we begin to use numbers that have a radix larger than 10: we must invent *new* symbols for the numerals greater than decimal 9. One popular method used to invent new numerals is to borrow other well-known *symbols from the alphabet.*

Alphabetic letters can be adapted for each new numeral, such as A for decimal 10, B for decimal 11, C for decimal 12, and so on until all the new numerals have a symbol. Number systems that use a radix greater than 10 have not proved useful to date, except one, known as *hexadecimal.*

Hexadecimal numbers have a radix of *16, with numerals equivalent to decimal number 0 to decimal number 15.* Hexadecimal numbers are useful for binary work because *one* hexadecimal numeral is equivalent to a 4-bit binary number. Hexadecimal numbers then, may be thought of as a type of binary shorthand. Each hexadecimal numeral can be replaced by its equivalent binary number, or the reverse. (Hexadecimal is also commonly referred to as *hex* by most programmers.)

Classification According to Memory devices

↓ just like a human brain

* It is used to store data and instruction.
* Computer memory is the storage space in Computer where data is to be processed and instructions required for processing are stored
* Two types of devices memory.
  1) Embedded memory microcontroller
  2) External memory microcontroller
     → digital cameras, mobile phone, music Players.

Embedded memory microcontroller :

* when an embedded system has a microcontroller unit that has all the functional blocks available on a chip is called an embbed microcontroller
* Embedded systems Control many devices in common use today.

for eg:
* data memory
* I/O ports
* Serial communication
* Counters & timers
* interrupts

* parallel
* device

on the chip is an embedded micro controller

Embedded RISC Processor: → Microcontroller with small instruction set are called RISC

* A major applic. area for microcontrollers are embedded system

* RISC → Reduced Instructions set Computer.

* It focus on reducing the number & complexity of instructions

* Reduces the number of cycles needed / instruction
* Simplify the addr. modes
* clock pulse 50 MHz to 200 MHz

Eg: MIPS, SPARC & ALPHA Processor reduced.

* There are two approaches of the designs of the control unit of a microprocessor.

1) Hardware Approaches.

2) Software "

* - The execute of an instruction a number of steps are required. For each step a number of control are are generated by the control unit of the processor.

* If executed each instructions if there is a separate electronic circuitary in a control unit, which produces all the necessary signals

* This approaches of the design of the control section is called RISC processor

* It is a hardware approach. It is also called hard-wired approach. Micro programming is not used

* If the instruction is more than this approach becomes complex

* So RISC processor, use a small number of simple instruction and a few addressing modes.

* They are faster the RISC processor each instruction is executed in to load & store instructions.

* For this purpose they employ a large number of general Purpose register.

(large memory is used).