The 8052 Tutorial & Reference

Version 1.00 – 25/Jun/2004

Created with RedHat Linux 9.0 and OpenOffice 1.0

Table of Contents

Welcome to the 8052.com Tutorial & Reference

Welcome to the **8052.com Tutorial & Reference**.

This document is, essentially, a compilation of all the tutorial information found on **8052.com**. It was assembled as a service to the 8052 community as a general offline reference of 8052 information recognizing that using the website as reference is not always practical or easy.

We decided to put together this document after receiving literally hundreds of requests for an offline version of the website that could be easily referenced or printed. While we could, perhaps, simply "zip up" the HTML files that make up the website, that doesn't really make it particularly easy to print or reference.

We would ask all those that find this document useful
to visit www.8052.com on a regular basis.

8052.com is able to continue operation due to the sponsorship of companies that present their products to the 8052 community by way of non-intrusive, non-popup banner ads on the website. Our sponsors' continued interest in **8052.com**, and subsequent support, is dependent on a significant percentage of **8052.com** visitors doing us the favor of clicking on our sponsors' links.

If this document is useful to you, please show your support by visiting www.8052.com periodically and clicking on those banner ads that are of interest to you. Doing so is what makes this document and the **8052.com** website possible.

In addition, you'll find the following features available to you at 8052.com:

1. **MESSAGE FORUM & COMMUNITY**. The public message forum (http://www.8052.com/forum) is an ideal place to discuss 8052-related questions and projects with other 8052 gurus. Not only are many experts ready to give you free advice online, the 55,000+ messages posted over the last five years provide a vast library in which you may find that your question has already been answered.

2. **ONLINE STORE**. 8052.com offers an Online Store where you may purchase products from many popular 8052 vendors, such as Keil, Phytec, Phyton, Raisonance, Rigel, and Vault Information Services. At the time of this writing, all products are subject to a 10% discount off of the normal price that these companies charge—making 8052.com the best place to acquire your 8052 SBCs, compilers, IDEs, and part programmers. You may visit the Online Store at http://www.8052.com/store.

3. **8052 NEWS**. Many 8052-product vendors submit their news releases directly to 8052.com. When you visit 8052.com, you have access to the latest breaking news in the 8052 market.

4. **8052 HOME PAGES**. 8052.com offers you a place to publish your 8052-related home page. With 8052.com Home Pages, you may publish your 8052-related software or hardware so that others in the 8052 community may benefit from your contribution. Of course, you may also browse and learn from other users' 8052 home pages.

Foreword

Despite it's relatively old age, the 8052 is one of the most popular microcontrollers in use today. Many derivative microcontrollers have since been developed that are based on--and compatible with--the 8052. Thus, the ability to program an 8052 is an important skill for anyone who plans to develop products that will take advantage of microcontrollers.

Many web pages, books, and tools are available to the 8052 developer.

I hope the information contained in this document will assist you in mastering 8052 programming. While it is not my intention that this document replace a traditional book purchased at your local book store, it is entirely possible that this may be the case. It is likely that this document contains everything you will need to learn 8052 assembly language programming. Of course, this document is free and you get what you pay for so if, after reading this document, you still are lost you may find it necessary to buy a book.

This document is both a tutorial and a reference tool. The various chapters of the document will explain the 8052 step by step. The chapters are targeted at people who are attempting to learn 8052 assembly language programming. The appendices are a useful reference tool that will assist both the novice programmer as well as the experienced professional developer.

This document assumes the following:

1. A general knowledge of programming.
2. An understanding of decimal, hexadecimal, and binary number systems.
3. A general knowledge of hardware.

That is to say, no knowledge of the 8052 is assumed--however, it is assumed you've done some amount of programming before, have a basic understanding of hardware, and a firm grasp on the three numbering systems mentioned above. The concept of converting a number from decimal to hexadecimal and/or to binary is not within the scope of this document--and if you can't do those types of conversions there are probably some concepts that will not be completely understandable.

This document attempts to address the need of the typical programmer. For example, there are certain features that are nifty and in some cases very useful--but 95% of the programmers will never use these features. To make this document more applicable to the general programming public some details may be skimmed over very briefly--or not at all.

In any case, I hope you find this document useful. If you have any questions, comments, suggestions, or corrections I welcome them at http://www.8052.com/contact.phtml (I used to provide a direct email, but have been forced to discontinue that practice due to excessive spamming).

Happy programming!

Craig Steiner (Author)

Chapter 1: An Introduction to Microcontrollers

Many developers new to microcontrollers (which the 8052 is) come from a PC/Windows or Macintosh environment. While most programming *concepts* will transfer over to the 8052 environment with no problem, there are some issues that a surprising number of people stumble upon as they enter the microcontroller world. Before delving into the details of microcontrollers and, specifically, the 8052, we'll address some common stumbling blocks.

1.1 What is a Microcontroller?

A **microcontroller** (often abbreviated **MCU**) is a single computer chip (integrated circuit) that executes a user program, normally for the purpose of *controlling some device*—hence the name micro*controller*.

The program is normally contained either in a second chip, called an EPROM, or within the same chip as the microcontroller itself. A microcontroller is normally found in devices such as microwave ovens, automobiles, keyboards, CD players, cell phones, VCRs, security systems, time & attendance clocks, etc.

Microcontrollers are used in devices that require some amount of computing power but don't require as much computing power as that provided by a complex (and expensive) 486 or Pentium system which generally requires a large amount of supporting circuitry (large motherboards, hundreds of megabytes of RAM, hard drives, hard drive controllers, video cards, etc). A microwave oven just doesn't need that much computing power.

Microcontroller-based systems are generally smaller, more reliable, and cheaper. They are ideal for the types of applications described above where *cost* and unit *size* are very important considerations. In such applications it is almost always desirable to produce circuits that require the smallest number of integrated circuits, that require the smallest amount of physical space, require the least amount of energy, and cost as little as possible.

The popular website HowStuffWorks.com has a good article that can provide you with some background information about microcontrollers in general at http://www.howstuffworks.com/microcontroller.htm.

1.1.1 Microcontroller Program Storage

The program for a microcontroller is normally stored on a memory integrated circuit (IC), called an EPROM, or on the microcontroller chip itself.

An **EPROM** (Electrically Programmable Read Only Memory) is a special type of integrated circuit that does nothing more than store program code or other data but which is maintained even when the power to the EPROM is turned off. Once you've developed software for a microcontroller it is normally programmed (or "burned") into an EPROM chip, and that chip is subsequently physically inserted into the circuitry of your hardware. The microcontroller accesses the program stored in the EPROM and executes it. Thus the program is made available to the microcontroller without the need for a hard drive, floppy drive, or any of the other circuitry necessary to access such devices.

In recent years, more and more microcontrollers offer the capability of having programs loaded internally into the microcontroller chip itself. Thus, rather than having a circuit that includes both a microcontroller and an external EPROM chip, it is now entirely possible to have a single microcontroller which stores the program code internally.

1.1.2 "Loading" your Microcontroller Program

The manner in which you transfer your software from your PC to your hardware depends on whether you are using an EPROM or are transferring the program directly to the microcontroller.

Programming an EPROM requires special hardware, called an **EPROM Programmer**. An EPROM programmer is a device that connects to your PC, via either the serial, parallel, or USB port. You then place the EPROM chip into a socket on the device and then use special software that transfers your program from the PC to the EPROM Programmer, which in turn "burns" your program into the chip. Once your program is burned into the EPROM you remove the EPROM and insert it in your circuit.

Programming a microcontroller that stores the program within the microcontroller itself generally requires a serial port be available for downloading updates to the program. Many of these devices have a "back door" in that you can still insert the microcontroller into an EPROM Programmer and load the software as described in the previous paragraph. If you are designing your circuit from scratch, however, it is often a good idea to plan for the possibility of programming the microcontroller without removing the IC from the circuit itself—this is especially true of surface-mount parts that may be difficult to remove from the circuit. The datasheet for the microcontroller you choose to use should provide you the information necessary to design your circuit for this capability.

1.2 What is an 8051 or 8052?

The **8052** is an 8-bit microcontroller originally developed by Intel in the late 1970s. It included an instruction set of 255 operation codes (**opcodes**), 32 input/output lines, three user-controllable timers, an integrated and automatic serial port, and 256 bytes of on-chip RAM. The **8051** is a very similar MCU but it has only two timers and 128 bytes of on-chip RAM.

The 8052 was designed such that control of the MCU and all input/output between the MCU and external devices is accomplished via **Special Function Registers** (**SFR**s). Each SFR has an address between 128 and 255. Additional functions can be added to new *derivative* MCUs by adding additional SFRs while remaining compatible with the original 8052. This allows the developer to use the same software development tools with any MCU that is "8052-compatible."

Intel wisely licensed their "8052 core" to other semiconductor firms. This allowed the 8052-architecture to become an industry-wide standard. Now, more than 20 years later, dozens of semiconductor companies produce microcontrollers that are based on the original 8052 core. The additional features that each semiconductor-firm offers in their MCUs are accessed by utilizing new SFRs in addition to the standard and original 8052-SFRs that are found in all 8052-compatible MCUs.

Warning: While most microcontrollers based on the 8052 core will include the standard SFRs from the 8052 core, some derivatives may only implement a subset of them. They may also change the function of some bits. This is generally not the case, but it is something to look out for when using derivative chips.

In this document, the term "**8052**" will refer to any MCU that is compatible with the original 8052. As a minimum it will support the 8052 instruction set, support the 8052's 26 SFRs, provide three user timers, and have at least 256 bytes of Internal RAM.

The term "**8051**" will refer to any 8052-compatible MCU that doesn't meet the specifications in the previous paragraph, rather mirroring Intel's 8051 microcontroller which was a more limited version of the 8052. As a minimum, an 8051 must support the 21 SFRs supported by the original 8051 and support the standard 8052 instruction set. Generally an 8051 will have two user timers and 128 bytes of RAM on-chip, although some designs have as little as 64 bytes of on-chip RAM.

1.2.1 Derivative Chips

The term "**derivative chip**", in this document, will refer to any 8051 or 8052-compatible MCU that is produced by any semiconductor firm. There are currently hundreds of derivative chips produced by dozens of semiconductor firms.

A derivative chip will generally (but not always) be able to execute a standard 8051 or 8052 program without modification. A derivative chip must be based on the 8052 instruction set and support the appropriate SFRs (at least 21 SFRs for an 8051 or 26 for an 8052).

Software development tools designed for the 8052 can always be used to develop software for any derivative chip as long as the programmer explicitly defines any new SFRs that are supported by the derivative chip that they are using.

1.3 Using Windows or Linux

A surprising number of people ask whether or not a microcontroller is Windows-compatible, or if they can load Linux on their microcontroller. These questions demonstrate a fundamental lack of understanding about what a microcontroller is.

No, a microcontroller cannot run Windows nor can it run Linux. Nor is a microcontroller Windows-compatible, per se. Does your microwave oven run Linux? Is your automobile Windows-compatible? These questions are as silly as asking whether a microcontroller runs Windows.

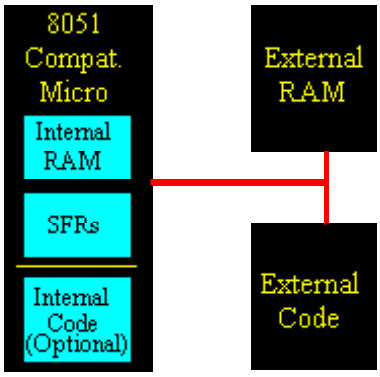
It is important to remember what microcontrollers are used for. A microcontroller is not a personal computer. It is an integrated circuit that will run your programs to control hardware devices such as those mentioned earlier. A microwave oven doesn't need Windows and your automobile certainly doesn't need Linux to maintain the correct air-fuel mixture.

That said, you can—and many developers do—use Windows or Linux to develop programs *for* microcontrollers. Many Windows products exist that allow you to write 8052 software in the Windows environment even though the software will ultimately be executed by a microcontroller. Once the software is executed by a microcontroller it doesn't matter whether it was originally developed under Windows, Linux, or any other operating system.

Chapter 2: Types of Memory

The 8052 has three very general types of memory. To program the 8052 effectively it is necessary to have a basic understanding of these memory types.

The memory types are illustrated in the following graphic. They are: *On-Chip Memory, Code Memory,* and *External RAM*.



Code Memory is code (or program) memory that is used to store the actual program. This often resides off-chip in the form of an EPROM. Many derivative chips allow program storage on the MCU itself (Internal Code Memory) and some modern derivative chips may not even support the concept of having code memory located off-chip.

External RAM is RAM memory that resides off-chip. This is often in the form of standard static RAM or flash RAM.

On-Chip Memory refers to any memory (Code, RAM, or other) that physically exists on the MCU itself. On-chip memory can be of several types, but we'll get to that shortly.

2.1 Code Memory

Code memory is the memory that holds the actual 8052 program that is to be run. This memory is conventionally limited to a maximum of 64K and comes in many shapes and sizes: Code memory may be found on-chip, either burned into the microcontroller as ROM or EPROM or loaded into flash program memory in newer derivatives. Code may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. Flash memory is another popular method of storing a program. Various combinations of these memory types may also be used--that is to say, it is possible to have 4K of code memory on-chip and 64k of code memory *off-chip* in an EPROM, depending on the derivative chip that you use.

When the program is stored on-chip, the 64K maximum is often reduced to 1k, 2k, 4k, 8k, 16k, or 32k. This varies depending on the derivative chip that is being used. Each derivative offers specific capabilities and one of the distinguishing factors from derivative to derivative is how much on-chip code memory the part has.

However, the code memory of a "classic" 8052 system is usually off-chip EPROM. This is especially true in low-cost development systems and in systems developed by students where learning how to interface with off-chip memory is part of the exercise.

Programming Tip: Since code memory is restricted to 64K, 8052 programs are limited to 64K. Some compilers offer ways to get around this limit when used with specially wired hardware and a technique known as "memory banking." However, without such special compilers and hardware, programs are limited to 64K.

Hardware Tip: Some manufacturers such as Dallas, Intel and Philips have special 8052 derivatives that can address several megabytes of memory.

2.2 External RAM

Although 8052s contain a small amount of on-chip RAM (see below), External RAM is also supported.

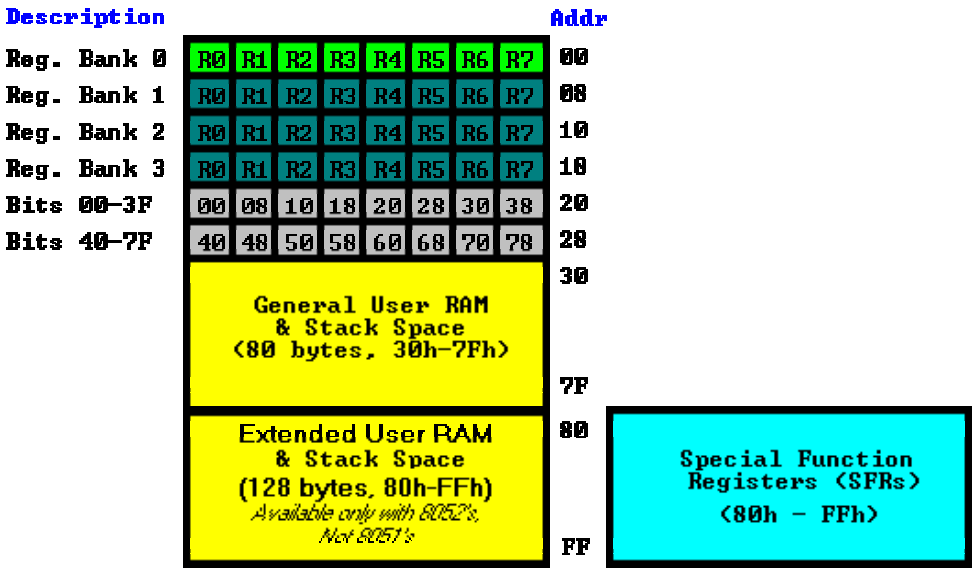
As the name suggests, External RAM is any random access memory that is found off-chip. Since the memory is off-chip the assembly language instructions to access it are slower and less flexible. For example, to increment an Internal RAM location by 1 requires only 1 instruction and 1 instruction cycle. To increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. In this case, external memory is 7 times slower and requires 4 times as much program memory!

What External RAM loses in speed and flexibility it gains in quantity. While Internal RAM is normally limited to 256 bytes (128 with 8051's), the 8052 supports External RAM up to 64K.

Programming Tip: The 8052 may only address 64k of RAM. To expand RAM beyond this limit requires programming and hardware tricks. You may have to do this "by hand" since many compilers and assemblers, while providing support for programs in excess of 64k, do not support more than 64k of RAM. This is rather strange since it has been my experience that programs can usually fit in 64k but often RAM is what is lacking. Thus if you need more than 64k of RAM, check to see if your compiler supports it-- but if it doesn't, be prepared to do it by hand.

2.3 On-Chip Memory

As mentioned at the beginning of this chapter, the 8052 includes a certain amount of on-chip memory. On-chip memory is really one of two types: **Internal RAM** and **Special Function Register** (SFR) memory. The layout of the 8052's internal memory is presented in the following memory map:



2.3.1 Internal RAM

As can be seen, the 8052 has a bank of 256 bytes of Internal RAM. This Internal RAM is found on-chip within the 8052 so it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying its contents. Internal RAM is volatile so when the 8052 is reset this memory is undefined.

The 256 bytes of Internal RAM are subdivided as shown in the memory map. The first 8 bytes (00h - 07h) are "register bank 0". By manipulating a certain SFR, a program may choose to use register banks 0, 1, 2, or 3. These alternative register banks are located in internal RAM in addresses 08h through 1Fh. We'll discuss "register banks" more in a later chapter. For now it is sufficient to know that they "live" in and are part of Internal RAM.

Bit Memory also lives in and is part of Internal RAM. We'll talk more about bit memory very shortly, but for now just keep in mind that bit memory actually resides in internal RAM, from addresses 20h through 2Fh.

Keep in mind that all of Internal RAM is byte-wide memory, regardless of whether it is used by register banks or bit memory. The Internal RAM allocated to register banks 1, 2, and 3 and bit memory may be used for your own use if you will not being using those register banks and/or bit variables.

The 208 bytes remaining of Internal RAM, from addresses 30h through FFh, may be used by user variables that need to be accessed frequently or at high-speed. This area is also utilized by the microcontroller as a storage area for the operating stack. This fact severely limits the 8052's stack since, as illustrated in the memory map, the area reserved for the stack is only 208 bytes--and usually it is less since these 208 bytes have to be shared between the stack and user variables.

Programming Note: While the 8052 has 256 bytes of Internal RAM, the 8051 only has 128 bytes of Internal RAM—further restricting the amount of RAM available on-chip for user variables and the operating stack.

2.3.1.2 The Stack

The stack is a “last in, first out” (LIFO) storage area that exists in Internal RAM. It is used by the 8052 to store values that the user program manually pushes onto the stack as well as to store the return addresses for CALLs and interrupt service routines (more on these topics later).

The stack is defined and controlled by a Special Function Register called the Stack Pointer, or SP. SP, as a standard 8-bit SFR, holds a value between 0 and 255 that represents the Internal RAM address of the *end* of the current stack. If a value is removed from the stack, it will be taken from the Internal RAM address pointed to by SP and SP will subsequently be decremented by 1. If a value is pushed onto the stack, SP will first be incremented and then the value will be inserted in Internal RAM at the address now pointed to by SP.

SP is initialized to 07h when an 8052 is first booted. This means the first value to be pushed onto the stack will be placed at Internal RAM address 08h (07h + 1), the second will be placed at 09h, etc.

Programming Tip: By default, the 8052 initializes the Stack Pointer (SP) to 07h when the microcontroller is booted. This means that the stack will start at address 08h and expand upwards. If you will be using the alternate register banks (banks 1, 2 or 3) you must initialize the stack pointer to an address above the highest register bank you will be using, otherwise the stack will overwrite your alternate register banks. Similarly, if you will be using bit variables it is usually a good idea to initialize the stack pointer to some value greater than 2Fh to guarantee that your bit variables are protected from the stack. We will talk about the register banks and Bit Memory below.

2.3.1.3 Register Banks

The 8052 uses 8 "R" registers which are used in many of its instructions. These "R" registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7) and are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the Accumulator, we would execute the following assembly language instruction:

```
ADD A,R4
```

Thus if the Accumulator (A) contained the value 6 and R4 contained the value 3, the Accumulator would contain the value 9 after this instruction was executed.

However, as the memory map shows, the "R" Register R4 is really part of Internal RAM. Specifically, R4 is address 04h of Internal RAM. This can be seen in the bright green section of the memory map. Thus the above instruction accomplishes the same thing as the following operation:

```
ADD A,04h
```

This instruction adds the value found in Internal RAM address 04h to the value of the Accumulator, leaving the result in the Accumulator. Since R4 is really Internal RAM address 04h, the above instruction effectively accomplishes the same thing as the previous ADD instruction.

But watch out! As the memory map shows, the 8052 has four distinct register banks. When the 8052 is first booted up register bank 0 (addresses 00h through 07h) is used by default. However, your program

may instruct the 8052 to use one of the alternate register banks; i.e., register banks 1, 2, or 3. In this case, R4 will no longer be the same as Internal RAM address 04h. For example, if your program instructs the 8052 to use register bank 1, register R4 will now be synonymous with Internal RAM address 0Ch. If you select register bank 2, R4 is synonymous with 14h, and if you select register bank 3 it is synonymous with address 1Ch.

The register bank is selected by setting or clearing the bits RS0 and RS1 in the Program Status Word (PSW) Special Function Register. For example:

```
MOV PSW,#00h        ;Sets register bank 0
MOV PSW,#08h        ;Sets register bank 1
MOV PSW,#10h        ;Sets register bank 2
MOV PSW,#18h        ;Sets register bank 3
```

The above instructions will make more sense after we cover the topics of Special Function Registers.

The concept of register banks adds a great level of flexibility to the 8052, especially when dealing with interrupts (we'll talk about interrupts later). However, always remember that the register banks really reside in the first 32 bytes of Internal RAM.

Programming Tip: If you only use the first register bank (i.e. bank 0), you may use Internal RAM locations 08h through 1Fh for your own use. If you plan to use register banks 1, 2, or 3, be very careful about using addresses below 20h as you may end up overwriting the value of "R" registers from other register banks.

2.3.1.4 Bit Memory

The 8052, being a communications and control-oriented microcontroller that often has to deal with “on” and “off” situations, gives the developer the ability to access a number of bit variables directly with simple instructions to set, clear, and compare these bits. These variables may be either 1 or 0.

There are 128 bit variables available to the developer, numbered 00h through 7Fh. The developer may make use of these variables with commands such as SETB and CLR. For example, to set bit number 24h (hex) to 1 you would execute the instruction:

```
SETB 24h
```

It is important to note that Bit Memory, like the Register Banks above, is really a part of Internal RAM. In fact, the 128 bit variables occupy the 16 bytes of Internal RAM from 20h through 2Fh. Thus, if you write the value FFh to Internal RAM address 20h you’ve effectively set bits 00h through 07h. That is to say that the instruction:

```
MOV 20h,#0FFh
```

is equivalent to the instructions:

```
SETB 00h
SETB 01h
SETB 02h
SETB 03h
SETB 04h
SETB 05h
SETB 06h
SETB 07h
```

As illustrated above, Bit Memory isn't really a new type of memory. It's really just a subset of Internal RAM. Since the 8052 provides special instructions to access these 16 bytes of memory on a bit-by-bit basis it is useful to think of it as a separate type of memory. But always keep in mind that it is just a subset of Internal RAM--and that operations performed on Internal RAM can change the values of the bit variables.

Programming Tip: If your program does not use bit variables you may use Internal RAM locations 20h through 2Fh for your own use. If you plan to use bit variables be very careful about using addresses from 20h through 2Fh as you may end up overwriting the value of your bits.

Programming Tip: By default, the 8052 initializes the Stack Pointer (SP) to 07h when the microcontroller is booted. This means that the stack will start at address 08h and expand upwards. If you will be using the alternate register banks (banks 1, 2 or 3) you must initialize the stack pointer to an address above the highest register bank you will be using, otherwise the stack will overwrite your alternate register banks. Similarly, if you will be using bit variables it is usually a good idea to initialize the stack pointer to some value greater than 2Fh to guarantee that your bit variables are protected from the stack.

While Bit Memory 00h through 7Fh are for developer-defined functions in their programs, Bit Memory 80h and above are used to access certain SFRs (see below) on a bit-by-bit basis. For example, if output lines P0.0 through P0.7 are all clear (0) and you want to turn on the P0.1 output line you may either execute:

```
MOV P0,#02h
```

or you may execute:

```
SETB 81h
```

Both of these instructions accomplish the same thing. However, using the SETB command will turn on the P0.0 line without affecting the status of any of the other P0 output lines. The MOV command effectively turns off all the other output lines which, in some cases, may not be acceptable.

When dealing with bit addresses of 80h and above remember that the bits refer to the bits of corresponding SFRs that are divisible by 8. This is a complicated way of saying that bits 80h through 87h refer to bits 0 through 7 of SFR 80h. Bits 88h through 8Fh refer to bits 0 through 7 of SFR 88h. Bits 90h through 97f refer to bits 0 through 7 of 90h, etc.

2.3.2 Special Function Register (SFR) Memory

Special Function Registers (**SFR**s) are areas of memory that control specific functionality of the 8052 MCU. For example, four SFRs permit access to the 8052's 32 input/output lines (8 lines per SFR). Another SFR allows a program to read or write to the 8052's serial port. Other SFRs allow the user to set the serial baud rate, control and access timers, and configure the 8052's interrupt system.

When programming, SFRs have the illusion of being Internal Memory. For example, if you want to write the value "1" to Internal RAM location 50h you would execute the instruction:

```
MOV 50h,#01h
```

Similarly, if you want to write the value "1" to the 8052's serial port you would write this value to the SBUF SFR, which has an SFR address of 99 Hex. Thus, to write the value "1" to the serial port you would execute the instruction:

```
MOV 99h,#01h
```

As you can see, it appears as if the SFR is part of Internal Memory. **This is not the case**. When using this method of memory access (it's called "direct address", more on that soon), any instruction that has an address of 00h through 7Fh refers to an Internal RAM memory address; any instruction with an address of 80h through FFh refers to an SFR control register.

Programming Tip #1: SFRs are used to control the way the 8052 functions. Each SFR has a specific purpose and format that will be discussed later. Not all addresses above 80h are assigned to SFRs. However, this area may NOT be used as additional RAM memory even if a given address has not been assigned to an SFR.

Programming Tip #2: Since direct access to addresses 80h through FFh refers to SFRs, direct access cannot be used to access Internal RAM addresses 80h through FFh. The upper 128 bytes of Internal RAM must be accessed using "Indirect Addressing" which will be explained in a subsequent chapter.

Chapter 3: Special Function Registers (SFRs)

The 8052 is a flexible microcontroller with a relatively large number of modes of operation. Your program may inspect and/or change the operating mode of the 8052 by manipulating the values of the **Special Function Registers** (SFRs).

SFRs are accessed as if they were normal Internal RAM. The only difference is that Internal RAM is addressed in "direct mode" with addresses 00h through 7Fh whereas SFR registers are accessed in the range of 80h through FFh.

Each SFR has an address (80h through FFh) and a name. The following chart provides a graphical presentation of the 8052's SFRs, their names, and their address.

80	P0	SP	DPL	DPH				PCON	87
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
90	P1								97
98	SCON	SBUF							9F
A0	P2								A7
A8	IE								AF
B0	P3								B7
B8	IP								B9
C0									C7
C8	T2CON		RCAP2L	RCAP2H	TL2	TH2			CF
D0	PSW								D7
D8									DF
E0	ACC								E7
E8									EF
F0	B								F7
F8									FF

Blue background are I/O port SFRs
Yellow background are control SFRs
Green background are other SFRs

As you can see, although the address range of 80h through FFh offer 128 possible addresses, there are only 26 SFRs in a standard 8052 (21 with an 8051). All other addresses in the SFR range (80h through FFh) are considered invalid. Writing to or reading from these registers may produce undefined values or behavior.

Programming Tip: It is recommended that you not read or write to SFR addresses that have not been assigned to an SFR. Doing so may provoke undefined behavior and may cause your program to be incompatible with other 8052-derivatives that use the given SFR for some other purpose.

3.1 Referencing SFRs

When writing code in assembly language, SFRs may be referenced either by their name or their address.

For example, we can see in the chart above that the SBUF SFR is at address 99h. If we wanted to write code that would write the value 24h to the SBUF SFR in assembly language, we could code it as:

```
MOV 99h,#24h
```

This instruction moves the value 24h into address 99h. Since the value 99h is in the range of 80h-FFh, it refers to an SFR. Further, since 99h refers to the SBUF SFR we know that this instruction will accomplish our goal of writing the value 24h to the SBUF SFR.

While the above instruction certainly works, it is not necessarily easy to remember the address of each SFR when you are writing software. Thus, all 8052 assemblers allow you to use the name of the SFR in code rather than its numeric address. The above instruction would more commonly be written as:

```
MOV SBUF,#24h
```

The instruction is much easier to read since it is obvious the SBUF SFR is being accessed. The assembler will automatically convert this to its numeric address at assemble-time.

3.2.1 Referencing Bits of SFRs

Individual bits of SFRs are referenced in one of two ways. The general convention is to name the SFR followed by a period and the bit number. For example, SCON.0 refers to bit 0 (the least significant bit) of the SCON SFR. SCON.7 refers to bit 7 (the most significant bit) of SCON.

These bits also have names: SCON.0 is RI and SCON.7 is SM0. It is also acceptable to refer to the bits by their name, although in this document we will usually refer to them in the SCON.0 format since that tells you which bit in which SFR you are dealing with.

3.3 Bit-Addressable SFRs

All SFRs that have addresses divisible by 8, such as 80h, 88h, 90h, 98h, etc. are bit-addressable. This means that you can set or clear individual bits of these SFRs using the SETB and CLR instruction.

Programming Tip: The SFRs whose names appear in red in the chart above are SFRs that may be accessed via bit operations; these also happen to be the first column of SFRs on the left side of the chart since the left-most column is divisible by 8. The other SFRs cannot be accessed using bit operations such as SETB or CLR.

3.3 SFR Types

As mentioned in the chart itself, the SFRs that have a blue background are SFRs related to the I/O ports. The 8052 has four I/O ports of 8 bits, for a total of 32 I/O lines. Whether a given I/O line is high or low and the value read from the line are controlled by the SFRs in green.

The SFRs with yellow backgrounds are SFRs that in some way control the operation or the configuration of some aspect of the 8052. For example, TCON controls the timers, SCON controls the serial port.

The remaining SFRs, with green backgrounds, are "other SFRs." These SFRs can be thought of as auxiliary SFRs in the sense that they don't directly configure the 8052 but obviously the 8052 cannot operate without them. For example, once the serial port has been configured using SCON, the program may read or write to the serial port using the SBUF register.

3.4 SFR Descriptions

This section will endeavor to quickly overview each of the standard SFRs found in the above SFR chart map. It is not the intention of this section to fully explain the functionality of each SFR--this information will be covered in separate chapters of the tutorial. This section is to just give you a general idea of what each SFR does.

P0 (Port 0, Address 80h, Bit-Addressable): This is input/output port 0. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 0 is pin P0.0, bit 7 is pin P0.7. Writing a value of 1 to a bit of this SFR will set a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

Programming Tip: While the 8051 has four I/O ports (P0, P1, P2, and P3), if your hardware uses external RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using external RAM chips) you may not use P0 or P2. This is because the 8052 uses ports P0 and P2 to address the external memory. Thus if you are using external RAM or code memory you may only use ports P1 and most of P3 for your own use.

SP (Stack Pointer, Address 81h): This is the stack pointer of the microcontroller. This SFR indicates where the next value to be taken from the stack will be read from in Internal RAM. If you push a value onto the stack, the value will be written to the address of SP + 1. That is to say, if SP holds the value 07h, a PUSH instruction will push the value onto the stack at address 08h. This SFR is modified by all instructions that modify the stack, such as PUSH, POP, LCALL, RET, RETI, and whenever interrupts are triggered by the microcontroller.

Programming Tip: The SP SFR, on startup, is initialized to 07h. This means the stack will start at 08h and start expanding upward in internal RAM. Since alternate register banks 1, 2, and 3 as well as the user bit variables occupy internal RAM from addresses 08h through 2Fh, it is necessary to initialize SP in your program to some other value if you will be using the alternate register banks and/or bit memory. It's not a bad idea to initialize SP to 2Fh as the first instruction of every one of your programs unless you are 100% sure you will not be using the register banks and bit variables.

DPL/DPH (Data Pointer Low/High, Addresses 82h/83h): The SFRs DPL and DPH work together to represent a 16-bit value called the Data Pointer. The data pointer is used in operations regarding external RAM and some instructions involving code memory. Since it is an unsigned two-byte integer value, it can represent values from 0000h to FFFFh (0 through 65,535 decimal).

Programming Tip: DPTR is really DPH and DPL taken together as a 16-bit value. In reality, you almost always have to deal with DPTR one byte at a time. For example, to push DPTR onto the stack you must first push DPL and then DPH. You can't simply push DPTR onto the stack. Additionally, there is an instruction to "increment DPTR." When you execute this instruction, the two bytes are operated upon as a 16-bit value. However, there is no instruction which decrements DPTR. If you wish to decrement the value of DPTR you must write your own code to do so. DPTR is a useful storage location for the occasional 16-bit value you are manipulating in your own code—especially if you frequently need to increment that value.

Code Library: You will find a "Decrement DPTR" source code at the 8052.com Code Library at **http://www.8052.com/codelib.phtml**.

PCON (Power Control, Addresses 87h): The Power Control SFR is used to control the 8052's power control modes. Certain operation modes of the 8052 allow the 8052 to go into a type of "sleep" mode that requires much less power. These modes of operation are controlled through PCON. Additionally, one of the bits in PCON is used to double the effective baud rate of the 8052's serial port.

TCON (Timer Control, Addresses 88h, Bit-Addressable): The Timer Control SFR is used to configure and modify the way in which the 8052's two basic timers operate. This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate that each timer has overflowed. Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags that are set when an external interrupt has occurred.

T2CON (Timer Control 2, Addresses C8h, Bit-Addressable): The Timer Control 2 SFR is used to configure and control the way in which timer 2 operates. This SFR is only available on 8052s, not on 8051s.

TMOD (Timer Mode, Addresses 89h): The Timer Mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit timer, an 8-bit auto-reload timer, a 13-bit timer, or two separate timers. Additionally, you may configure the timers to only count when an external pin is activated or to count "events" that are indicated on an external pin.

TL0/TH0 (Timer 0 Low/High, Addresses 8Ah/8Bh): These two SFRs, taken together, represent timer 0. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

TL1/TH1 (Timer 1 Low/High, Addresses 8Ch/8Dh): These two SFRs, taken together, represent timer 1. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

TL2/TH2 (Timer 2 Low/High, Addresses CCh/CDh): These two SFRs, taken together, represent timer 2. Their exact behavior depends on how the timer is configured in the T2CON SFR.

RCAP2L/RCAP2H (Timer 2 Capture Low/High, Addresses CAh/CBh): These two SFRs, taken together, represent the timer 2 "capture" register. It may be used as a reload value for timer 2, or to capture the value of timer 2 under certain circumstances. The exact purpose and function of these two SFRs depends on the configuration of T2CON.

P1 (Port 1, Address 90h, Bit-Addressable): This is input/output port 1. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 1 is pin P1.0, bit 7 is pin P1.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

SCON (Serial Control, Addresses 98h, Bit-Addressable): The Serial Control SFR is used to configure the behavior of the 8052's on-board serial port. This SFR controls how the baud rate of the serial port is determined, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

Programming Tip: To use the 8052's on-board serial port, it is generally necessary to initialize the following SFRs: SCON, TCON, and TMOD. This is because SCON controls the serial port but in most cases the program will wish to use one of the timers to establish the serial port's baud rate. In this case, it is necessary to configure timer 1 or timer 2 by initializing TCON and TMOD, or T2CON.

SBUF (Serial Control, Addresses 99h): The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin. Likewise, any value which the 8052 receives via the serial port's RXD pin will be delivered to the user program via SBUF. In other words, SBUF acts as if it were really two registers, one which serves as the output port when written to and the other which serves as the input port when read from.

P2 (Port 2, Address A0h, Bit-Addressable): This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

Programming Tip: While the 8052 has four I/O port (P0, P1, P2, and P3), if your hardware uses external RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using external RAM chips) you may not use P0 or P2. This is because the 8052 uses ports P0 and P2 to address the external memory. Thus if you are using external RAM or code memory you may only use ports P1 and P3 for your own use.

IE (Interrupt Enable, Addresses A8h): The Interrupt Enable SFR is used to enable and disable specific interrupts. The low 7 bits of the SFR are used to enable/disable the specific interrupts, where as the highest bit is used to enable or disable ALL interrupts. Thus, if the high bit of IE is 0 all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

P3 (Port 3, Address B0h, Bit-Addressable): This is input/output port 3. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

IP (Interrupt Priority, Addresses B8h, Bit-Addressable): The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8052, an interrupt may either be of low (0) priority or high (1) priority. An interrupt may only interrupt interrupts of lower priority. For example, if we configure the 8052 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing. However, if a serial interrupt is executing no other interrupt will be able to interrupt the serial interrupt routine since the serial interrupt routine has the highest priority.

PSW (Program Status Word, Addresses D0h, Bit-Addressable): The Program Status Word is used to store a number of important bits that are set and cleared by 8052 instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW register contains the register bank select flags that are used to select which of the "R" register banks are currently selected.

Programming Tip: If you write an interrupt handler routine, it is a very good idea to always save the PSW SFR on the stack and restore it when your interrupt is complete. Many 8052 instructions modify the bits of PSW. If your interrupt routine does not guarantee that PSW is the same upon exit as it was upon entry your program is bound to behave rather erratically and unpredictably--and it will be tricky to debug since the behavior will tend not to make any sense.

ACC (Accumulator, Addresses E0h, Bit-Addressable): The Accumulator is one of the most-used SFRs on the 8052 since it is involved in so many instructions. The Accumulator resides as an SFR at E0h which means the instruction MOV A,#20h is the same as MOV E0h,#20h. However, it is a good idea to use the first method since it only requires two bytes whereas the second option requires three bytes.

B (B Register, Addresses F0h, Bit-Addressable): The "B" register is implicitly used in two instructions: the multiply and divide operations. The B register is also commonly used by programmers as an auxiliary register to store temporarily values.

3.5 Other SFRs

The chart at the beginning of this chapter is a summary of all the SFRs that exist in a standard 8052. All derivative microcontrollers of the 8052 must support these basic SFRs in order to maintain compatibility with the underlying MCS-51 standard.

A common practice when semiconductor firms wish to develop a new 8052 derivative is to add additional SFRs to support new functions that exist in the new chip. For example, the Dallas Semiconductor DS80C320 is upward compatible with the 8052. This means that any program that runs on a standard 8052 should run without modification on the DS80C320. This also means that all the SFRs defined above also apply to the Dallas component.

However, since the DS80C320 provides many new features that the standard 8052 does not, there must be some way to control and access these new features. This is accomplished by adding additional SFRs to those listed here. For example, since the DS80C320 supports two serial ports (as opposed to just one on the 8052), the SFRs SBUF2 and SCON2 have been added. In addition to all the SFRs listed above, the DS80C320 also recognizes these two new SFRs as valid and uses their values to determine the mode of operation of the secondary serial port. Obviously, these new SFRs have been assigned to SFR addresses that were unused in the original 8052. In this manner, new 8052 derivative chips may be developed which will run existing 8052 programs.

Programming Tip: If you write a program that utilizes new SFRs that are specific to a given derivative chip and not included in the above SFR list, your program will not run properly on a standard 8052 where that SFR does not exist. Thus, only use non-standard SFRs if you are sure that your program will only have to run on that specific microcontroller. Likewise, if you write code that uses non-standard SFRs and subsequently share it with a third-party, be sure to let that party know that your code is using non-standard SFRs to save them the headache of realizing that due to strange behavior at run-time.

Chapter 4: Basic Registers

A number of 8052 registers can be considered “basic.” Very little can be done without them and a detailed explanation of each one is warranted to make sure the reader understands these registers before getting into more complicated areas of development.

4.1 The Accumulator

If you’ve worked with any other assembly language you will be familiar with the concept of an accumulator register.

The Accumulator, as its name suggests, is used as a general register to accumulate the results of a large number of instructions. It can hold an 8-bit (1-byte) value and is the most versatile register the 8052 has due to the sheer number of instructions that make use of the accumulator. More than half of the 8052’s 255 instructions manipulate or use the Accumulator in some way.

For example, if you want to add the number 10 and 20, the resulting 30 will be stored in the Accumulator. Once you have a value in the Accumulator you may continue processing the value or you may store it in another register or in memory.

4.2 The "R" registers

The "R" registers are sets of eight registers that are named R0, R1, through R7.

These registers are used as auxiliary registers in many operations. To continue with the above example, perhaps you are adding 10 and 20. The original number 10 may be stored in the Accumulator whereas the value 20 may be stored in, say, register R4. To process the addition you would execute the command:

```
ADD A,R4
```

After executing this instruction the Accumulator will contain the value 30.

You may think of the "R" registers as very important auxiliary, or "helper", registers. The Accumulator alone would not be very useful if it were not for these "R" registers.

The "R" registers are also used to store values temporarily. For example, let’s say you want to add the values in R1 and R2 together and then subtract the values of R3 and R4. One way to do this would be:

```
MOV A,R3      ;Move the value of R3 into the accumulator
ADD A,R4      ;Add the value of R4
MOV R5,A      ;Store the resulting value temporarily in R5
MOV A,R1      ;Move the value of R1 into the accumulator
ADD A,R2      ;Add the value of R2
SUBB A,R5     ;Subtract the value of R5 (which now contains R3 + R4)
```

As you can see, we used R5 to temporarily hold the sum of R3 and R4. Of course, this isn’t the most efficient way to calculate (R1+R2) - (R3 +R4) but it does illustrate the use of the "R" registers as a way to store values temporarily.

As mentioned earlier, there are four sets of “R” registers—register bank 0, 1, 2, and 3. When the 8052 is first powered up, register bank 0 (addresses 00h through 07h) is used by default. In this case, for example, R4 is the same as Internal RAM address 04h. However, your program may instruct the 8052 to use one of the alternate register banks; i.e., register banks 1, 2, or 3. In this case, R4 will no longer be the same as Internal RAM address 04h. For example, if your program instructs the 8052 to use register bank 1, register R4 will now be synonymous with Internal RAM address 0Ch. If you select register bank 2, R4 is synonymous with 14h, and if you select register bank 3 it is synonymous with address 1Ch.

The concept of register banks adds a great level of flexibility to the 8052, especially when dealing with interrupts (we'll talk about interrupts later). However, always remember that the register banks really reside in the first 32 bytes of Internal RAM.

4.3 The "B" Register

The "B" register is very similar to the Accumulator in the sense that it may hold an 8-bit (1-byte) value.

The "B" register is only used implicitly by two 8052 instructions: MUL AB and DIV AB. Thus, if you want to quickly and easily multiply or divide A by another number, you may store the other number in "B" and make use of these two instructions.

Aside from the MUL and DIV instructions, the "B" register is often used as yet another temporary storage register much like a ninth "R" register.

4.4 The Program Counter (PC)

The Program Counter (PC) is a 2-byte address that tells the 8052 where the next instruction to execute is found in memory. When the 8052 is initialized PC always starts at 0000h and is incremented each time an instruction is executed. It is important to note that PC isn’t always incremented by one. Since some instructions are 2 or 3 bytes in length the PC will be incremented by 2 or 3 in these cases.

The Program Counter is special in that there is no way to directly modify its value. That is to say, you can’t do something like PC=2430h. On the other hand, if you execute LJMP 2430h you’ve effectively accomplished the same thing.

It is also interesting to note that while you may change the value of PC (by executing a jump instruction, etc.) there is no way to read the value of PC. That is to say, there is no way to ask the 8052 "What address are you about to execute?" As it turns out, this is not completely true: There is one trick that may be used to determine the current value of PC. This trick will be covered in a later chapter.

4.5 The Data Pointer (DPTR)

The Data Pointer (DPTR) is the 8052’s only user-accessible 16-bit (2-byte) register. The Accumulator, "R" registers, and "B" register are all 1-byte values. The PC just described is a 16-bit value but isn’t directly user-accessible as a working register.

DPTR, as the name suggests, is used to point to data. It is used by a number of commands that allow the 8052 to access external memory. When the 8052 accesses external memory it accesses the memory at the address indicated by DPTR.

While DPTR is most often used to point to data in external memory or code memory, many developers take advantage of the fact that it's the only true 16-bit register available. It is often used to store 2-byte values that have nothing to do with memory locations.

4.6 The Stack Pointer (SP)

The Stack Pointer, like all registers except DPTR and PC, may hold an 8-bit (1-byte) value. The Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from.

When you push a value onto the stack, the 8052 first increments the value of SP and then stores the value at the resulting memory location.

When you pop a value off the stack, the 8052 returns the value from the memory location indicated by SP, and then decrements the value of SP.

This order of operation is important. When the 8052 is initialized SP will be initialized to 07h. If you immediately push a value onto the stack, the value will be stored in Internal RAM address 08h. This makes sense taking into account what was mentioned two paragraphs above: First the 8051 will increment the value of SP (from 07h to 08h) and then will store the pushed value at that memory address (08h).

SP is modified directly by the 8052 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered (more on interrupts later. Don't worry about them for now!).

Chapter 5: Addressing Modes

As is the case with all microcomputers from the PDP-8 onwards, the 8052 utilizes several memory addressing modes. An "addressing mode" refers to how you are accessing ("addressing") a given memory location or data value. In summary, the addressing modes are listed below with an example of each:

Immediate Addressing	MOV A,#20h
Direct Addressing	MOV A,30h
Indirect Addressing	MOV A,@R0
External Direct	MOVX A,@DPTR
External Indirect	MOVX A,@R0
Code Indirect	MOVC A,@A+DPTR

Each of these addressing modes provides important flexibility to the programmer.

5.1 Immediate Addressing

Immediate addressing is so-named because the value to be stored in memory immediately follows the opcode in memory. That is to say, the instruction itself dictates what value will be stored in memory.

For example:

```
MOV A,#20h
```

This instruction uses Immediate Addressing because the Accumulator (A) will be loaded with the value that immediately follows; in this case 20h (hexadecimal).

Immediate Addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible. It is used to load the same, known value every time the instruction executes.

5.2 Direct Addressing

Direct addressing is so-named because the value to be stored in memory is obtained by directly retrieving it from another memory location.

For example:

```
MOV A,30h
```

This instruction will read the data out of Internal RAM address 30h (hexadecimal) and store it in the Accumulator (A).

Direct addressing is generally fast since, although the value to be loaded isn't included in the instruction, it is quickly accessible since it is stored in the 8052's Internal RAM. It is also much more flexible than Immediate Addressing since the value to be loaded is whatever is found at the given address--which may change.

Also, it is important to note that when using direct addressing any instruction that refers to an address between 00h and 7Fh is referring to Internal RAM. Any instruction that refers to an address between 80h and FFh is referring to the SFR control registers that control the 8052 itself.

The obvious question that may arise is, "If direct addressing an address from 80h through FFh refers to SFRs, how can I access the upper 128 bytes of Internal RAM that are available with the 8052?" The answer is: You can't access them using direct addressing. As stated, if you directly refer to an address of 80h through FFh you will be referring to an SFR.

However, you may access the 8052's upper 128 bytes of RAM by using the next addressing mode, "indirect addressing."

5.3 Indirect Addressing

Indirect addressing is a very powerful addressing mode that in many cases provides an exceptional level of flexibility. Indirect addressing is also the only way to access the upper 128 bytes of Internal RAM found on an 8052.

Indirect addressing appears as follows:

```
MOV A,@R0
```

This instruction causes the 8052 to analyze the value of the R0 register. The 8052 will then load the Accumulator (A) with the value from Internal RAM that is found at the address indicated by R0.

For example, let's suppose R0 holds the value 40h and Internal RAM address 40h holds the value 67h. When the above instruction is executed the 8052 will check the value of R0. Since R0 holds 40h the 8052 will get the value out of Internal RAM address 40h (which holds 67h) and store it in the Accumulator. Thus, the Accumulator ends up holding 67h.

Indirect addressing always refers to Internal RAM; *it never refers to an SFR*. In a prior example we mentioned that SFR 99h can be used to write a value to the serial port. Thus one may think that the following would be a valid solution to write the value '1' to the serial port:

```
MOV R0,#99h  ;Load the address of the serial port
MOV @R0,#01h        ;Send 01 to the serial port -- WRONG!!
```

This is not valid. Since indirect addressing *always* refers to Internal RAM these two instructions would write the value 01h to Internal RAM address 99h on an 8052.

On an 8051 these two instructions would produce an undefined result since the 8051 only has 128 bytes of Internal RAM.

5.4 External Direct

External memory is accessed using a suite of instructions that use what I call "External Direct" addressing. I call it this because the address to be accessed is contained directly in the DPTR register, but it accesses external memory.

There are only two commands that use External Direct addressing mode:

```
MOVX A,@DPTR
MOVX @DPTR,A
```

As you can see, both commands utilize DPTR. In these instructions, DPTR must first be loaded with the address of external memory that you wish to read or write. Once DPTR holds the correct external memory address, the first command will move the contents of that external memory address into the Accumulator.

For example, to read the contents of external RAM address 1516h, we'd execute the instructions:

```
MOV DPTR,#1516h     ;Select the external address to read
MOVX A,@DPTR        ;Move the contents of external RAM into accumulator
```

The second command will do the opposite: it will allow you to write the value of the Accumulator to the external memory address pointed to by DPTR. For example, to write the contents of the Accumulator to external RAM address 1516 we'd execute the instructions:

```
MOV DPTR,#1516h     ;Select the external address to read
MOVX @DPTR,A        ;Move the contents of external RAM into accumulator
```

Programming Tip: Technically, accessing external memory with the MOVX @DPTR instructions is *indirect* addressing since the address to be accessed is referred to indirectly by the DPTR register. However, to directly access a specific external RAM memory location the most direct way is to load DPTR with the address in question and access it with the MOVX instruction. Thus while this approach is technically "indirect," I called it "External Direct" since it is the most direct method of accessing a specific external RAM memory location and to differentiate it from the following addressing mode (External Indirect) which is very similar to the "indirect addressing" mentioned in section 5.3 in its use of R0 and R1.

5.5 External Indirect

External memory can also be accessed using a form of indirect addressing that I call External Indirect. This form of addressing is usually only used in relatively small projects that have a very small amount of external RAM. An example of this addressing mode is:

```
MOVX @R0,A
```

Once again, the value of R0 is first read and the value of the Accumulator is written to that address in External RAM. Since the value of @R0 can only be 00h through FFh the project would effectively be limited to 256 bytes of External RAM. There are relatively simple hardware/software tricks that can be implemented to access more than 256 bytes of memory using External Indirect addressing; however, it is usually easier to use External Direct addressing if your project has more than 256 bytes of External RAM.

5.6 Code Indirect

Two additional 8052 instructions allow the developer to access the program code itself. This is useful for accessing data tables, strings, etc. The two instructions are:

```
MOVC A,@A+DPTR
MOVC A,@A+PC
```

For example, if we wanted to access the data stored in code memory at address 2021h, we'd execute the instructions:

```
MOV DPTR,#2021h     ;Set DPTR to 2021h
CLR A               ;Clear the accumulator (set to 00h)
MOVC A,@A+DPTR      ;Read code memory address 2021h into the accumulator
```

The MOVC A,@A+DPTR moves the value contained in the code memory address that is pointed to by adding DPTR to the Accumulator.

Chapter 6: Program Flow

When an 8052 is first initialized the PC SFR is reset to 0000h. The 8052 then begins to execute instructions sequentially in memory unless a program instruction causes the PC to be otherwise altered. There are various instructions that can modify the value of the PC; specifically, conditional branching instructions, direct jumps and calls, and "returns" from subroutines. Additionally, interrupts, when enabled, can cause the program flow to deviate from its otherwise sequential scheme.

6.1 Conditional Branching

The 8052 contains a suite of instructions which, as a group, are referred to as "conditional branching" instructions. These instructions cause program execution to follow a non-sequential path if a certain condition is true. Take, for example, the JB instruction. This instruction means "Jump if Bit Set." An example of the JB instruction might be:

```
        JB 45h,HELLO
        NOP
HELLO:....
```

In this case, the 8052 will analyze the contents of bit 45h. If the bit is set program execution will jump immediately to the label HELLO, skipping the NOP instruction. If the bit is not set the conditional branch fails and program execution continues, as usual, with the NOP instruction that follows.

Conditional branching is the fundamental building block of program logic since all "decisions" are accomplished by using conditional branching. Conditional branching can be thought of as the "IF... THEN" structure in 8052 assembly language.

An important note worth mentioning about conditional branching is that the program may only branch to instructions located within 128 bytes prior to or 127 bytes after the address that follows the conditional branch instruction. This means that in the above example the label HELLO must be within +/- 128 bytes of the memory address that contains the conditional branching instruction.

6.2 Direct Jumps

While conditional branching is extremely important, it is often necessary to make a direct branch to a given memory location without basing it on a given logical decision. This is equivalent to saying "GOTO" in BASIC. In this case you want the program flow to continue at a given memory address without considering any conditions.

This is accomplished in the 8052 using "Direct Jump and Call" instructions. As illustrated in the last paragraph, this suite of instructions causes program flow to change unconditionally.

Consider the example:

```
        LJMP NEW_ADDRESS
             .
             .
             .
NEW_ADDRESS: ....
```

The LJMP instruction in this example means "Long Jump." When the 8052 executes this instruction the PC is loaded with the address of NEW_ADDRESS and program execution continues sequentially from there.

The obvious difference between the Direct Jump and Call instructions and the conditional branching is that with Direct Jumps and Calls program flow always changes. With conditional branching program flow only changes if a certain condition is true.

It is worth mentioning that, aside from LJMP, there are two other instructions that cause a direct jump to occur: the SJMP and AJMP commands. Functionally, these two commands perform the exact same function as the LJMP command--that is to say, they always cause program flow to continue at the address indicated by the command. However, these instructions differ from LJMP in that they are not capable of jumping to any address. They both have limitations as to the "range" of the jumps.

1. The SJMP command, like the conditional branching instructions, can only jump to an address within +/- 128 bytes of the SJMP command.
2. The AJMP command can only jump to an address that is in the same 2k block of memory as the AJMP command. That is to say, if the AJMP command is at code memory location 650h, it can only do a jump to addresses 0000h through 07FFh (0 through 2047, decimal).

You may ask yourself, "Why would I want to use the SJMP or AJMP command which have restrictions as to how far they can jump if they do the same thing as the LJMP command which can jump anywhere in memory?" The answer is simple: The LJMP command requires three bytes of code memory whereas both the SJMP and AJMP commands require only two. If you are developing an application that has memory restrictions you can often save quite a bit of memory using the 2-byte AJMP/SJMP instructions instead of the 3-byte instruction.

Recently, I wrote a program that required 2100 bytes of memory but I had a memory restriction of 2k (2048 bytes). I did a search/replace changing all LJMPs to AJMPs and the program shrunk downto 1950 bytes. Without changing any logic whatsoever in my program I saved 150 bytes and was able to meet my 2048 byte memory restriction.

NOTE: Some assemblers will do the above conversion for you automatically. That is, they'll automatically change your LJMPs to SJMPs whenever possible. This is a nifty and very powerful capability that you may want to look for in an assembler if you plan to develop projects that have relatively tight memory restrictions.

6.3 Direct Calls

Another operation that will be familiar to seasoned programmers is the LCALL instruction. This is similar to a "GOSUB" command in Basic.

When the 8052 executes an LCALL instruction it immediately pushes the current Program Counter onto the stack and then continues executing code at the address indicated by the LCALL instruction.

Similar in format to the AJMP instruction that was described in the previous section, the ACALL instruction provides a way to perform the equivalent of an "LCALL" with a 2-byte instruction (instead of 3) as long as the target routine is within the same 2k block of memory.

6.4 Returns from Routines

Another structure that can cause program flow to change is the "Return from Subroutine" instruction, known as RET in 8051 Assembly Language. The RET instruction, when executed, returns to the address following the instruction that called the given subroutine. More accurately, it returns to the address that is stored on the stack.

The RET command is direct in the sense that it always changes program flow without basing it on a condition, but is variable in the sense that where program flow continues can be different each time the RET instruction is executed depending on from where the subroutine was called originally.

6.5 Interrupts

An interrupt is a special feature that allows the 8052 to break from its normal program flow to execute an immediate task, providing the illusion of "multi-tasking." The word "interrupt" can often be substituted with the word "event."

An interrupt is triggered whenever a corresponding event occurs. When the event occurs, the 8052 temporarily puts "on hold" the normal execution of the main program and executes a special section of code referred to as the "Interrupt Service Routine" (ISR). The ISR performs whatever special functions are required to handle the event and then returns control to the 8052 at which point program execution continues as if it had never been interrupted.

The topic of interrupts is somewhat tricky and very important. For that reason, an entire chapter will be dedicated to the topic.

Chapter 7: Instruction Set, Timing, and Low-Level Information

In order to understand--and better make use of--the 8052, it is necessary to understand some underlying information concerning timing.

The 8052 operates with timing derived from an external crystal or a clock signal generated by some other system. A crystal is a component that allows an electronic oscillator to run at a very precisely known frequency. One can find crystals of virtually any frequency depending on the application requirements. When using an 8052, one of the most common crystal frequencies is 11.0592 megahertz.

Why would anyone pick such an oddball frequency? There's a reason which has to do with generating baud rates and we'll talk more about it in the Serial Communication chapter. For the remainder of this discussion we'll assume that we're using an 11.0592Mhz crystal.

Microcontrollers (and many other electrical systems) use their oscillators to synchronize operations. The 8052 uses its crystal or clock for precisely that: to synchronize its internal operation. The 8052 operates using what are called "instruction cycles." A single instruction cycle is the minimum amount of time in which a single 8052 instruction can be executed, although many instructions take multiple cycles.

A cycle is, in reality, 12 clock cycles from the crystal. That is to say, if an instruction takes one instruction cycle to execute, it will take 12 clocks of the crystal to execute. Since we know the crystal oscillates 11,059,200 times per second and that one instruction cycle is 12 clock cycles, we can calculate how many instruction cycles the 8052 can execute per second:

11,059,200 / 12 = 921,600

This means that the 8052 can execute 921,600 single-cycle instructions per second. Since a large number of 8052 instructions are single-cycle instructions it is often considered that the 8051 can execute roughly 1 million instructions per second (MIPS), although in reality it is less--and, depending on the instructions being used, an estimate of about 600,000 instructions per second is more realistic.

For example, if you are using exclusively 2-cycle instructions you would find that the 8052 executes 460,800 instructions per second. The traditional 8052 also has two really slow instructions (MUL AB and DIV AB) that require a full 4 cycles to execute--if you were to execute nothing but those instructions you'd find performance to be about 230,400 instructions per second.

It is again important to emphasize that not all instructions execute in the same amount of time. The fastest instructions require one instruction cycle (12 clock cycles), many others require two instruction cycles (24 clock cycles), and the two very slow math operations require four instruction cycles (48 clock cycles).

NOTE: Many derivative chips change instruction timing. For example, many optimized versions of the 8052 execute instructions in 4 oscillator cycles instead of 12; such a chip would be effectively 3 times faster than the 8052 when used with the same 11.0592 MHz crystal.

Since all the instructions require different amounts of time to execute, a very obvious question comes to mind: How can one keep track of time in a time-critical application if we have no reference to time in the outside world? Luckily, the 8052 includes timers which allow us to time events with high precision--which is the topic of the next chapter.

Chapter 8: Timers

The 8052 comes equipped with three timers, all of which may be controlled, set, read, and configured individually. The timers have three general functions: 1) Keeping time and/or calculating the amount of time between events, 2) Counting the events themselves, or 3) Generating baud rates for the serial port.

The three timer uses are distinct so we will talk about each of them separately. The first two uses will be discussed in this chapter while the use of timers for baud rate generation will be discussed in the chapter relating to serial ports.

Programming Tip: While the 8052 has three timers, the 8051 only has two. Timers 0 and 1 work and are configured the same and will be discussed first. Timer 2, available only on 8052-compatible derivatives, is configured differently and will be discussed later in this chapter.

8.1 How does a timer count?

The answer to this question is very simple: A timer always counts up. It doesn't matter whether the timer is being used as a timer, a counter, or a baud rate generator: A timer is always incremented by the microcontroller.

Programming Tip: Some derivative chips actually allow the program to configure whether the timers count up or down. However, since this option only exists on some derivatives it is beyond the scope of this tutorial which is aimed at the standard 8052. It is only mentioned here in the event that you absolutely need a timer to count backwards, you will know that you may be able to find an 8052-compatible microcontroller that does it.

8.2 Using Timers to Measure Time

Obviously, one of the primary uses of timers is to measure time. We will discuss this use of timers first and will subsequently discuss the use of timers to count events. When a timer is used to measure time it is also called an "interval timer" since it is measuring the time of the interval between two events.

8.2.1 How long does a timer take to count?

First, it's worth mentioning that when a timer is in interval timer mode (as opposed to event counter mode) and correctly configured, it will increment by 1 every machine cycle. As you will recall from the previous chapter, a single machine cycle consists of 12 crystal pulses. Thus a running timer will be incremented:

11,059,200 / 12 = 921,600 times per second

Unlike instructions--some of which require 1 instruction cycle, others 2, and others 4--the timers are consistent: They will always be incremented once per instruction cycle. Thus if a timer has counted from 0 to 50,000 you may calculate:

50,000 / 921,600 = .0543 seconds

.0543 seconds have passed. In plain English, about half of a tenth of a second, or one-twentieth of a second.

Obviously it's not very useful to know .0543 seconds have passed. If you want to execute an event once per second you'd have to wait for the timer to count from 0 to 50,000 18.45 times. How can you wait "half of a time?" You can't. So we come to another important calculation.

Let's say we want to know how many times the timer will be incremented in .05 seconds. We can do simple multiplication:

.05 * 921,600 = 46,080 cycles

This tells us that it will take .05 seconds (1/20th of a second) to count from 0 to 46,080.

Obviously, this is a little more useful. If you know it takes 1/20th of a second to count from 0 to 46,080 and you want to execute some event every second you simply wait for the timer to count from 0 to 46,080 twenty times; then you execute your event, reset the timers, and wait for the timer to count up another 20 times. In this manner you will effectively execute your event once per second, accurate to within thousandths of a second.

Thus, we now have a system with which to measure time. All we need to review is how to control the timers and initialize them to provide us with the information we need.

8.2.2 Timer SFRs

As mentioned before, the 8052 has two timers which each function essentially the same way. One timer is TIMER0 and the other is TIMER1. The two timers share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to maintaining the value of the timer itself (TH0/TL0 and TH1/TL1).

The SFRs used to control and manipulate the timers are presented in the following table.

SFR Name	Description	SFR Address	Bit Addressable ?
TH0	Timer 0 High Byte	8Ch	No
TL0	Timer 0 Low Byte	8Ah	No
TH1	Timer 1 High Byte	8Dh	No
TL1	Timer 1 Low Byte	8Bh	No
TCON	Timer Control	88h	Yes
TMOD	Timer Mode	89h	No

Timer 0 has two SFRs dedicated exclusively to itself: TH0 and TL0. TL0 is the low-byte of the value of the timer while TH0 is the high-byte of the value of the timer. That is to say, when Timer 0 has a value of 0, both TH0 and TL0 will contain 0. When Timer 0 has the value 1000 (decimal), TH0 will hold the high byte of the value (3 decimal) and TL0 will contain the low byte of the value (232 decimal). Reviewing low/high byte notation, recall that you must multiply the high byte by 256 and add the low byte to calculate the final value. In this case:

(TH0 * 256) + TL0 = 1000
(3 * 256) + 232 = 1000

Timer 1 works the exact same way, but its SFRs are TH1 and TL1.

Since there are only two bytes devoted to the value of each timer it is apparent that the maximum value a timer may have is 65,535. If a timer contains the value 65,535 and is subsequently incremented, it will reset--or overflow--back to 0.

8.2.3 TMOD SFR

Let's first talk about our first control SFR: TMOD (Timer Mode). The TMOD SFR is used to control the mode of operation of both timers. Each bit of the SFR gives the microcontroller specific information concerning how to run a timer. The high four bits (bits 4 through 7) relate to Timer 1 whereas the low four bits (bits 0 through 3) perform the exact same functions, but for timer 0.

The individual bits of TMOD have the following functions:

TMOD (89h) SFR

Bit	Name	Explanation of Function	Timer
7	GATE1	When this bit is set the timer will only run when INT1 (P3.3) is high. When this bit is clear the timer will run regardless of the state of INT1.	1
6	C/T1	When this bit is set the timer will count events on T1 (P3.5). When this bit is clear the timer will be incremented every machine cycle.	1
5	T1M1	Timer mode bit (see below)	1
4	T1M0	Timer mode bit (see below)	1
3	GATE0	When this bit is set the timer will only run when INT0 (P3.2) is high. When this bit is clear the timer will run regardless of the state of INT0.	0
2	C/T0	When this bit is set the timer will count events on T0 (P3.4). When this bit is clear the timer will be incremented every machine cycle.	0
1	T0M1	Timer mode bit (see below)	0
0	T0M0	Timer mode bit (see below)	0

As you can see in the above chart, four bits (two for each timer) are used to specify a mode of operation. The modes of operation are:

TIMER MODES

TxM1	TxM0	Timer Mode	Description of Timer Mode
0	0	0	13-bit Timer
0	1	1	16-bit Timer
1	0	2	8-bit auto-reload
1	1	3	Split timer mode

8.2.3.1 13-bit Time Mode (mode 0)

Timer mode "0" is a 13-bit timer. This is a relic that was kept around in the 8052 to maintain compatibility with its predecessor, the 8048. The 13-bit timer mode is not normally used in new development.

When the timer is in 13-bit mode, TLx will count from 0 to 31. When TLx is incremented from 31, it will "reset" to 0 and increment THx. Thus, effectively, only 13 bits of the two timer bytes are being used: bits 0-4 of TLx and bits 0-7 of THx. This also means, in essence, the timer can only contain 8192 values. If you set a 13-bit timer to 0, it will overflow back to zero 8192 instruction cycles later.

There really is very little reason to use this mode and it is only mentioned so you won't be surprised if you ever end up analyzing archaic code that has been passed down through the generations (a generation in a programming shop is often on the order of about 3 or 4 months!).

8.2.3.2 16-bit Time Mode (mode 1)

Timer mode "1" is a 16-bit timer. This is a very commonly used mode. It functions just like 13-bit mode except that all 16 bits are used.

TLx is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536 distinct values. If you set a 16-bit timer to 0, it will overflow back to 0 after 65,536 machine cycles.

8.2.3.3 8-bit Time Mode (mode 2)

Timer mode "2" is an 8-bit auto-reload mode. What is that, you may ask? Simple. When a timer is in mode 2, THx holds the "reload value" and TLx is the timer itself. TLx starts counting up. When TLx reaches 255 and is subsequently incremented instead of resetting to 0 (as in the case of modes 0 and 1) it will be reset to the value stored in THx.

For example, let's say TH0 holds the value FDh and TL0 holds the value FEh. If we were to watch the values of TH0 and TL0 for a few machine cycles this is what we'd see:

Machine Cycle	TH0 Value	TL0 Value
1	FDh	Feh
2	FDh	FFh
3	FDh	FDh
4	FDh	Feh
5	FDh	FFh
6	FDh	FDh
7	FDh	Feh

As you can see, the value of TH0 never changed. In fact, when you use mode 2 you almost always set THx to a known value and TLx is the SFR that is constantly incremented. THx is initialized once, then left unchanged.

The benefit of auto-reload mode is that, perhaps, you want the timer to always have a value from 200 to 255. If you use mode 0 or 1, you'd have to check in code to see if the timer had overflowed and, if so,

reset the timer to 200. This takes precious instructions of execution time to check the value and/or to reload it.

When you use mode 2 the microcontroller takes care of this for you. Once you've configured a timer in mode 2 you don't have to worry about checking to see if the timer has overflowed nor do you have to worry about resetting the value--the microcontroller hardware will do it all for you.

The auto-reload mode is very commonly used for establishing a baud rate which we will talk more about in the Serial Communications chapter.

8.2.3.4 Split Timer Mode (mode 3)

Timer mode "3" is a split-timer mode. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. That is to say, Timer 0 is TL0 and Timer 1 is TH0. Both timers count from 0 to 255 and overflow back to 0. All the bits that are related to Timer 1 will now be tied to TH0 and all the bits related to Timer 0 will be tied to TL0.

While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into modes 0, 1 or 2 normally--however, you may not start or stop the real timer 1 since the bits that do that are now linked to TH0. The real timer 1, in this case, will be incremented every machine cycle no matter what.

The only real use I can see of using split timer mode is if you need to have two separate timers and, additionally, a baud rate generator. In such case you can use the real Timer 1 as a baud rate generator and use TH0/TL0 as two separate timers.

8.2.4 TCON SFR

Finally, there's one more SFR that controls the two timers and provides valuable information about them. The TCON SFR has the following structure:

TCON (88h) SFR

Bit	Name	Bit Address	Explanation of Function	Timer
7	TF1	8Fh	Timer 1 Overflow. This bit is set by the microcontroller when Timer 1 overflows.	1
6	TR1	8Eh	Timer 1 Run. When this bit is set Timer 1 is turned on. When this bit is clear Timer 1 is off.	1
5	TF0	8Dh	Timer 0 Overflow. This bit is set by the microcontroller when Timer 0 overflows.	0
4	TR0	8Ch	Timer 0 Run. When this bit is set Timer 0 is turned on. When this bit is clear Timer 0 is off.	0

As you may notice, we've only defined 4 of the 8 bits. That's because the other 4 bits of the SFR don't have anything to do with timers--they have to do with interrupts and they will be discussed in the chapter that addresses interrupts.

A new piece of information in this chart is the column "bit address." This is because this SFR is "bit-addressable." This means that if you want to set the bit TF1--which is the highest bit of TCON--you could execute the command:

```
MOV TCON, #80h
```

Since the SFR is bit-addressable you could just execute the command:

```
SETB TF1
```

This has the benefit of setting the high bit of TCON without changing the value of any of the other bits of the SFR. Usually when you start or stop a timer you don't want to modify the other values in TCON, so you take advantage of the fact that the SFR is bit-addressable.

8.2.5 Initializing a Timer

Now that we've discussed the timer-related SFRs we are ready to write code that will initialize the timer and start it running. As you'll recall, we first must decide what mode we want the timer to be in. In this case we want a 16-bit timer that runs continuously; that is to say, it is not dependent on any external pins.

We must first initialize the TMOD SFR. Since we are working with timer 0 we will be using the low 4 bits of TMOD. The first two bits, GATE0 and C/T0 are both 0 since we want the timer to be independent of the external pins. 16-bit mode is timer mode 1 so we must clear T0M1 and set T0M0. Effectively, the only bit we want to turn on is bit 0 of TMOD. Thus to initialize the timer we execute the instruction:

```
MOV TMOD,#01h
```

Timer 0 is now in 16-bit timer mode. However, the timer is not running. To start the timer running we must set the TR0 bit. We can do that by executing the instruction:

```
SETB TR0
```

Upon executing these two instructions timer 0 will immediately begin counting, being incremented once every instruction cycle (every 12 crystal pulses).

8.2.6 Reading the Timer

There are two common ways of reading the value of a 16-bit timer; which you use depends on your specific application. You may either read the actual value of the timer as a 16-bit number, or you may simply detect when the timer has overflowed.

8.2.6.1 Reading the value of a Timer

If your timer is in an 8-bit mode--that is, either 8-bit auto-reload mode or in split timer mode--then reading the value of the timer is simple. You simply read the 1-byte value of the timer and you're done.

However, if you're dealing with a 13-bit or 16-bit timer the chore is a little more complicated. Consider what would happen if you read the low byte of the timer as 255, then read the high byte of the timer as 15. In this case, what actually happened was that the timer value was 14/255 (high byte 14, low byte 255) but you read 15/255.

Why? Because you read the low byte as 255. But when you executed the next instruction a small amount of time passed--but enough for the timer to increment again at which time the value rolled over from 14/255 to 15/0. But in the process you've read the timer as being 15/255 instead of 14/255. Obviously there's a problem there.

The solution isn't that complicated, really. You read the high byte of the timer, then read the low byte, then read the high byte again. If the high byte read the second time is not the same as the high byte read the first time you repeat the cycle. In code, this would appear as:

```
REPEAT:
       MOV A,TH0
       MOV R0,TL0
       CJNE A,TH0,REPEAT
       ...
```

In this case, we load the accumulator with the high byte of Timer 0. We then load R0 with the low byte of Timer 0. Finally, we check to see if the high byte we read out of Timer 0--which is now stored in the Accumulator--is the same as the current Timer 0 high byte. If it isn't it means we've just "rolled over" and must re-read the timer's value--which we do by going back to REPEAT. When the loop exits we will have the low byte of the timer in R0 and the high byte in the Accumulator.

Another much simpler alternative is to simply turn off the timer run bit (i.e. CLR TR0), read the timer value, and then turn on the timer run bit (i.e. SETB TR0). In that case, the timer isn't running so no special tricks are necessary. Of course, this implies that your timer will be stopped for a few instruction cycles. Whether or not this is tolerable depends on your specific application.

8.2.6.2 Detecting Timer Overflow

Often it is only necessary to know that the timer has overflowed to 0. That is to say, you are not particularly interested in the value of the timer but rather you are interested in knowing when the timer has overflowed back to 0.

Whenever a timer overflows from its highest value back to 0, the microcontroller automatically sets the TFx bit in the TCON register. This is useful since rather than checking the exact value of the timer you can just check if the TFx bit is set. If the TF0 bit is set it means that timer 0 has overflowed; if TF1 is set it means that timer 1 has overflowed.

We can use this approach to cause the program to execute a fixed delay. As you'll recall, we calculated earlier that it takes the 8052 1/20th of a second to count from 0 to 46,080. However, the TFx flag is set when the timer overflows back to 0.

Thus, if we want to use the TFx flag to indicate when 1/20th of a second has passed we must set the timer initially to 65536 less 46080, or 19,456. If we set the timer to 19,456, 1/20th of a second later the timer will overflow. Thus we come up with the following code to execute a pause of 1/20th of a second:

```
MOV TH0,#76   ;High byte of 19,456 (76 * 256 = 19,456)
MOV TL0,#00   ;Low byte of 19,456 (19,456 + 0  = 19,456)
MOV TMOD,#01 ;Put Timer 0 in 16-bit mode
CLR TF0       ;Make sure TF0 bit is clear initially
SETB TR0      ;Make Timer 0 start counting
JNB TF0,$     ;If TF0 is not set, jump back to this same instruction
```

In the above code the first two lines initialize the Timer 0 starting value to 19,456. The next two instructions configure timer 0 and turn it on. Finally, the last instruction JNB TF0,$, reads "Jump back to the same instruction if TF0 is not set." The "$" operand means, in most assemblers, the address of the current instruction.

As long as the timer has not overflowed and the TF0 bit has not been set the program will keep executing this same instruction. After 1/20th of a second timer 0 will overflow, set the TF0 bit, and program execution will then break out of the loop.

8.2.7 Timing the length of events

The 8052 provides another useful toy that can be used to time the length of events.

For example, let's say we're trying to save electricity in the office and we're interested in how long a light is turned on each day. When the light is turned on, we want to measure time. When the light is turned off, we don't. One option would be to connect the light switch to one of the pins, constantly read the pin, and turn the timer on or off based on the state of that pin. While this would work fine, the 8052 provides us with an easier method of accomplishing this.

Looking again at the TMOD SFR, there is a bit called GATE0. So far we've always cleared this bit because we wanted the timer to run regardless of the state of the external pins. However, now it would be nice if an external pin could control whether the timer was running or not. It can.

All we need to do is connect the light switch to pin INT0 (P3.2) on the 8052 and set the bit GATE0. When GATE0 is set, Timer 0 will only run if P3.2 is high. When P3.2 is low (i.e., the light switch is off) the timer will automatically be stopped.

Thus, with no control code whatsoever, the external pin P3.2 can control whether or not our timer is running or not.

8.3 Using Timers as Event Counters

We've discussed how a timer can be used for the obvious purpose of keeping track of time. However, the 8052 also allows us to use the timers to count events.

This can be useful in many applications. Let's say you had a sensor placed across a road that would send a pulse every time a car passed over it. This could be used to determine the volume of traffic on the road. We could attach this sensor to one of the 8052's I/O lines and constantly monitor it, detect when it pulsed high, and then increment our counter when it went back to a low state. This is not terribly difficult, but requires some code. Let's say we hooked the sensor to P1.0; the code to count cars passing would look something like this:

```
JNB P1.0,$    ;If a car hasn't raised the signal, keep waiting
JB P1.0,$     ;The line is high, car is on the sensor right now
INC COUNTER   ;The car has passed completely, so we count it
```

As you can see, it's only three lines of code. But what if you need to be doing other processing at the same time? You can't be stuck in the JNB P1.0,$ loop waiting for a car to pass if you need to be doing other things. And what if you're doing other things when a car passes over? It's possible that the car will raise the signal and the signal will fall low again before your program checks the line status; this would result in the car not being counted. Of course, there are ways to get around even this limitation but the code quickly becomes big, complex, and ugly.

Luckily, since the 8052 provides us with a way to use the timers to count events we don't have to bother with it. It's actually painfully easy. We only have to configure one additional bit.

Let's say we want to use Timer 0 to count the number of cars that pass. If you look back to the bit table for the TCON SFR you will see there is a bit called "C/T0"--it's bit 2 (TCON.2). Reviewing the explanation of the bit we see that if the bit is clear then timer 0 will be incremented every instruction cycle. This is what we've already used to measure time.

If we set C/T0, however, timer 0 will monitor the P3.4 line. Instead of being incremented every machine cycle, timer 0 will count events on the P3.4 line. So in our case we simply connect our sensor to P3.4 and let the 8052 do the work. Then, when we want to know how many cars have passed, we just read the value of timer 0--the value of timer 0 will be the number of cars that have passed.

So what exactly is an event? What does timer 0 actually "count?" Speaking at the electrical level, the 8052 counts 1-0 transitions on the P3.4 line. This means that when a car first runs over our sensor it will raise the input to a high ("1") condition. At that point the 8052 will not count anything since this is a 0-1 transition. However, when the car has passed the sensor will fall back to a low ("0") state. This is a 1-0 transition and at that instant the counter will be incremented by 1.

It is important to note that the 8052 checks the P3.4 line each instruction cycle (12 clock cycles). This means that if P3.4 is low, goes high, and goes back low in 6 clock cycles it will probably not be detected by the 8052. This also means the 8052 event counter is only capable of counting events that occur at a maximum of 1/24th the rate of the crystal frequency. That is to say, if the crystal frequency is 12.000 Mhz it can count a maximum of 500,000 events per second (12.000 Mhz * 1/24 = 500,000). If the event being counted occurs more than 500,000 times per second it will not be able to be accurately counted by the 8052 without using additional external circuitry, a faster crystal, or a derivative chip that implements a machine cycle that uses less than 12 crystal cycles.

8.4. Using Timer 2

The 8052 has a third timer, Timer 2. Keep in mind that the 8051 and some lower-end derivative chips don't include Timer 2.

Timer 2 functions slightly different than Timers 0 and 1 and, for that reason, we are addressing this third timer separately from the first two.

8.4.1 T2CON SFR

The operation of Timer 2 (T2) is controlled almost entirely by the T2CON SFR, at address C8h. Note that since this SFR is evenly divisible by 8 that it is bit-addressable.

TCON (88h) SFR

Bit	Name	Bit Address	Explanation of Function
7	TF2	CFh	Timer 2 Overflow. This bit is set when T2 overflows. When T2 interrupt is enabled, this bit will cause the interrupt to be triggered. This bit will not be set if either TCLK or RCLK bits are set.
6	EXF2	Ceh	Timer 2 External Flag. Set by a reload or capture caused by a 1-0 transition on T2EX (P1.1), but only when EXEN2 is set. When T2 interrupt is enabled, this bit will cause the interrupt to be triggered.
5	RCLK	CDh	Timer 2 Receive Clock. When this bit is set, Timer 2 will be used to determine the serial port receive baud rate. When clear, Timer 1 will be used.
4	TCLK	CCh	Timer 2 Receive Clock. When this bit is set, Timer 2 will be used to determine the serial port transmit baud rate. When clear, Timer 1 will be used.
3	EXEN 2	CBh	Timer 2 External Enable. When set, a 1-0 transition on T2EX (P1.1) will cause a capture or reload to occur.
2	TR2	Cah	Timer 2 Run. When set, timer 2 will be turned on. Otherwise, it is turned off.
1	C/T2	C9h	Timer 2 Counter/Interval Timer. If clear, Timer 2 is an interval counter. If set, Timer 2 is incremented by 1-0 transition on T2 (P1.0).
0	CP/RL 2	C8h	Timer 2 Capture/Reload. If clear, auto reload occurs on timer 2 overflow, or T2EX 1-0 transition if EXEN2 is set. If set, a capture will occur on a 1-0 transition of T2EX if EXEN2 is set.

8.4.2 Timer 2 in Auto-Reload Mode

The first mode in which Timer 2 may be used is Auto-Reload. The auto-reload mode functions just like Timer 0 and Timer 1 in auto-reload mode, except that the Timer 2 auto-reload mode performs a full 16-bit reload (recall that Timer 0 and Timer 1 only have 8-bit reload values). When a reload occurs, the value of TH2 will be reloaded with the value contained in RCAP2H and the value of TL2 will be reloaded with the value contained in RCAP2L.

To operate Timer 2 in auto-reload mode, the CP/RL2 bit (T2CON.0) must be clear. In this mode, Timer 2 (TH2/TL2) will be reloaded with the reload value (RCAP2H/RCAP2L) whenever Timer 2 overflows; that is to say, whenever Timer 2 overflows from FFFFh back to 0000h. An overflow of Timer 2 will cause the TF2 bit to be set, which will cause an interrupt to be triggered, if Timer 2 interrupt is enabled. Note that TF2 will not be set on an overflow condition if either RCLK or TCLK (T2CON.5 or T2CON.4) are set.

Additionally, by also setting EXEN2 (T2CON.3), a reload will also occur whenever a 1-0 transition is detected on T2EX (P1.1). A reload that occurs as a result of such a transition will cause the EXF2 (T2CON.6) flag to be set, triggering a Timer 2 interrupt if said interrupt has been enabled.

8.4.3 Timer 2 in Capture Mode

A new mode, specific to Timer 2, is called "Capture Mode." As the name implies, this mode captures the value of Timer 2 (TH2 and TL2) into the capture SFRs (RCAP2H and RCAP2L). To put Timer 2 in capture mode, CP/RL2 (T2CON.0) must be set, as must be EXEN2 (T2CON.3).

When configured as mentioned above, a capture will occur whenever a 1-0 transition is detected on T2EX (P1.1). At the moment the transition is detected, the current values of TH2 and TL2 will be copied into RCAP2H and RCAP2L, respectively. At the same time, the EXF2 (T2CON.6) bit will be set, which will trigger an interrupt if Timer 2 interrupt is enabled.

Programming Tip 1: Note that even in capture mode, an overflow of Timer 2 will result in TF2 being set and an interrupt being triggered.

Programming Tip 2: Capture mode is an efficient way to measure the time between events. At the moment that an event occurs the current value of Timer 2 will be copied into RCAP2H/L. However, Timer 2 will not stop and an interrupt will be triggered. Thus your interrupt routine may copy the value of RCAP2H/L to a temporary holding variable without having to stop Timer 2. When another capture occurs, your interrupt can take the difference of the two values to determine the time transpired. Again, the main advantage is that you don't have to stop timer 2 to read its value, as is the case with timer 0 and timer 1.

8.4.4 Timer 2 as a Baud Rate Generator

Timer 2 may be used as a baud rate generator. This is accomplished by setting either RCLK (T2CON.5) or TCLK (T2CON.4). With an 8051, Timer 1 is the only timer that may be used to determine the baud rate of the serial port. Additionally, the receive and transmit baud rate must be the same.

With the 8052, however, the user may configure the serial port to receive at one baud rate and transmit with another. For example, if RCLK is set and TCLK is cleared, serial data will be received at the baud rate determined by Timer 2 whereas the baud rate of transmitted data will be determined by Timer 1.

Determining the auto-reload values for a specific baud rate is discussed in Serial Port Operation; the only difference is that in the case of Timer 2, the auto-reload value is placed in RCAP2H and RCAP2L, and the value is a 16-bit value rather than an 8-bit value.

NOTE: When Timer 2 is used as a baud rate generator (either TCLK or RCLK are set), the Timer 2 Overflow Flag (TF2) will not be set.

Chapter 9: Serial Communication

One of the 8052's many powerful features is its integrated UART, otherwise known as a serial port. The fact that the 8052 has an integrated serial port means that you may very easily read and write values to the serial port. If it were not for the integrated serial port, writing a byte to a serial line would be a rather tedious process requiring turning on and off one of the I/O lines in rapid succession to properly "clock out" each individual bit, including start bits, stop bits, and parity bits.

However, we do not have to do this. Instead, we simply need to configure the serial port's operation mode and baud rate. Once configured, all we have to do is write to an SFR to write a value to the serial port or read the same SFR to read a value from the serial port. The 8052 will automatically let us know when it has finished sending the character we wrote and will also let us know whenever it has received a byte so that we can process it. We do not have to worry about transmission at the bit level--which saves us quite a bit of coding and processing time.

9.1 Setting the Serial Port Mode

The first thing we must do when using the 8052's integrated serial port is, obviously, configure it. This lets us tell the 8052 how many data bits we want, the baud rate we will be using, and how the baud rate will be determined.

First, let's present the "Serial Control" (SCON) SFR and define what each bit of the SFR represents:

SCON (98h) SFR

Bit	Name	Bit Address	Explanation of Function
7	SM0	9Fh	Serial port mode bit 0
6	SM1	9Eh	Serial port mode bit 1.
5	SM2	9Dh	Mutliprocessor Communications Enable (explained later)
4	REN	9Ch	Receiver Enable. This bit must be set in order to receive characters.
3	TB8	9Bh	Transmit bit 8. The 9th bit to transmit in mode 2 and 3.
2	RB8	9Ah	Receive bit 8. The 9th bit received in mode 2 and 3.
1	TI	99h	Transmit Flag. Set when a byte has been completely transmitted.
0	RI	98h	Receive Flag. Set when a byte has been completely received.

Additionally, it is necessary to define the function of SM0 and SM1 by an additional table:

SM0	SM1	Serial Mode	Explanation	Baud Rate
0	0	0	8-bit Shift Register	Oscillator / 12
0	1	1	8-bit UART	Set by Timer 1 (*)
1	0	2	9-bit UART	Oscillator / 32 (*)
1	1	3	9-bit UART	Set by Timer 1 (*)

(*) **NOTE**: The baud rate indicated in this table is doubled if PCON.7 (SMOD) is set.

The SCON SFR allows us to configure the Serial Port. Thus, we'll go through each bit and review its function.

The high four bits (bits 4 through 7) are configuration bits.

Bits **SM0** and **SM1** let us set the serial mode to a value between 0 and 3, inclusive. The four modes are defined in the chart immediately above. As you can see, selecting the Serial Mode selects the mode of operation (8-bit/9-bit, UART or Shift Register) and also determines how the baud rate will be calculated.

In modes 0 and 2 the baud rate is fixed based on the oscillator's frequency. In modes 1 and 3 the baud rate is variable based on how often Timer 1 overflows. We'll talk more about the various Serial Modes in a moment.

The bit **SM2** is a flag for "Multiprocessor communication." Generally, whenever a byte has been received the 8052 will set the "RI" (Receive Interrupt) flag. This lets the program know that a byte has been received and that it needs to be processed. However, when SM2 is set the "RI" flag will only be triggered if the 9th bit received was a "1". That is to say, if SM2 is set and a byte is received whose 9th bit is clear, the RI flag will never be set. This can be useful in certain advanced serial applications that allow multiple 8052s (or other hardware) to communicate amongst themselves. For now it is safe to say that you will almost always want to clear this bit so that the flag is set upon reception of any character.

The bit **REN** is "Receiver Enable." This bit is very straightforward: If you want to receive data via the serial port, set this bit. You will almost always want to set this bit since leaving it cleared will prevent the 8052 from receiving serial data.

The low four bits (bits 0 through 3) are operational bits. They are used when actually sending and receiving data--they are not used to configure the serial port.

The **TB8** bit is used in modes 2 and 3. In modes 2 and 3, a total of nine data bits are transmitted. The first 8 data bits are the 8 bits of the main value, and the ninth bit is taken from TB8. If TB8 is set and a value is written to the serial port, the data's bits will be written to the serial line followed by a "set" ninth bit. If TB8 is clear the ninth bit will be "clear."

The **RB8** bit also operates in modes 2 and 3 and functions essentially the same way as TB8, but on the reception side. When a byte is received in modes 2 or 3, a total of nine bits are received. In this case, the first eight bits received are the data of the serial byte received and the value of the ninth bit received will be placed in RB8.

TI means "Transmit Interrupt." When a program writes a value to the serial port, a certain amount of time will pass before the individual bits of the byte are "clocked out" the serial port. If the program were to write another byte to the serial port before the first byte was completely output, the data being sent would be garbled. Thus, the 8052 lets the program know that it has "clocked out" the last byte by setting the TI bit. When the TI bit is set, the program may assume that the serial port is "free" and ready to send the next byte.

Finally, the **RI** bit means "Receive Interrupt." It functions similarly to the "TI" bit, but it indicates that a byte has been received. That is to say, whenever the 8052 has received a complete byte it will trigger the RI bit to let the program know that it needs to read the value quickly, before another byte is read.

NOTE: The TI bit is actually set halfway through the transmission of the stop bit, and the RI bit is actually set halfway through the reception of the incoming stop bit. This is normally not of major concern but if you are using a RS-485 network or some other network where you must specifically enable and disable the receiver and/or transmitter, you should be aware of the fact that when the RI/TI bit is set that slightly more time must pass before you disable the receiver/transmitter. Turning off the transmitter/receiver immediately will normally cause transmission errors.

9.2 Setting the Serial Port Baud Rate

Once the Serial Port Mode has been configured, as explained above, the program must configure the serial port's baud rate. This only applies to Serial Port modes 1 and 3. In modes 0 and 2, the baud rate is determined based on the oscillator's frequency and no additional configuration is necessary. In mode 0, the baud rate is always the oscillator frequency divided by 12. This means if you're crystal is 11.0592Mhz, mode 0 baud rate will always be 921,600 baud. In mode 2 the baud rate is always the oscillator frequency divided by 64, so a 11.0592Mhz crystal speed will yield a baud rate of 172,800.

In modes 1 and 3, the baud rate is determined by how frequently timer 1 overflows. The more frequently timer 1 overflows, the higher the baud rate. There are many ways one can cause timer 1 to overflow at a rate that determines a baud rate but the most common method is to put timer 1 in 8-bit auto-reload mode (timer mode 2) and set a reload value (TH1) that causes Timer 1 to overflow at a frequency appropriate to generate a baud rate.

To determine the value that must be placed in TH1 to generate a given baud rate, we may use the following equation (assuming PCON.7 is clear).

TH1 = 256 - ((Crystal / 384) / Baud)

If PCON.7 is set then the baud rate is effectively doubled, thus the equation becomes:

TH1 = 256 - ((Crystal / 192) / Baud)

For example, if we have an 11.059Mhz crystal and we want to configure the serial port to 19,200 baud we try plugging it in the first equation:

TH1 = 256 - ((Crystal / 384) / Baud)
TH1 = 256 - ((11059000 / 384) / 19200)
TH1 = 256 - ((28,799) / 19200)
TH1 = 256 - 1.5 = 254.5

As you can see, to obtain 19,200 baud with an 11.059Mhz crystal we'd have to set TH1 to 254.5. If we set it to 254 we will have achieved 14,400 baud and if we set it to 255 we will have achieved 28,800 baud. Thus we're stuck... But not quite... to achieve 19,200 baud we simply need to set PCON.7 (SMOD). When we do this we double the baud rate and utilize the second equation mentioned above. Thus we have:

TH1 = 256 - ((Crystal / 192) / Baud)
TH1 = 256 - ((11059000 / 192) / 19200)
TH1 = 256 - ((57699) / 19200)
TH1 = 256 - 3 = 253

Here we are able to calculate a nice, even TH1 value. Therefore, to obtain 19,200 baud with an 11.059MHz crystal we must:

1. Configure Serial Port mode 1 or 3 (for 8-bit or 9-bit serial mode).
2. Configure Timer 1 to timer mode 2 (8-bit auto-reload).
3. Set TH1 to 253 to reflect the correct frequency for 19,200 baud.
4. Set PCON.7 (SMOD) to double the baud rate.

9.3 Writing to the Serial Port

Once the serial port has been properly configured as explained above, the serial port is ready to be used to send and receive data. If you thought that configuring the serial port was simple, using the serial port will be a breeze.

To write a byte to the serial port one must simply write the value to the SBUF (99h) SFR. For example, if you wanted to send the letter "A" to the serial port, it could be accomplished as easily as:

```
MOV SBUF,#'A'
```

Upon execution of the above instruction the 8052 will begin transmitting the character via the serial port. Obviously transmission is not instantaneous--it takes a measurable amount of time to transmit the 8 data bits that make up the byte, along with its start and stop bits. And since the 8052 does not have a serial output buffer we need to be sure that a character is completely transmitted before we try to transmit the next character.

The 8052 lets us know when it is done transmitting a character by setting the TI bit in SCON. When this bit is set we know that the last character has been transmitted and that we may send the next character, if any. Consider the following code segment:

```
CLR TI       ;Be sure the bit is initially clear
MOV SBUF,#'A'       ;Send the letter 'A' to the serial port
JNB TI,$     ;Pause until the RI bit is set.
```

The above three instructions will successfully transmit a character and wait for the TI bit to be set before continuing. The last instruction says "Jump if the TI bit is not set to $". The "$" character, in most assemblers, means "the same address of the current instruction." Thus the 8052 will pause on the JNB instruction until the TI bit is set by the 8052 upon successful transmission of the character.

9.4 Reading the Serial Port

Reading data received by the serial port is equally easy. To read a byte from the serial port one just needs to read the value stored in the SBUF (99h) SFR after the 8052 has automatically set the RI flag in SCON.

For example, if your program wants to wait for a character to be received and subsequently read it into the Accumulator, the following code segment may be used:

```
JNB RI,$     ;Wait for the 8051 to set the RI flag
MOV A,SBUF   ;Read the character from the serial port
CLR RI       ;Clear the 8051 receive flag
```

The first line of the above code segment waits for the 8052 to set the RI flag; again, the 8052 sets the RI flag automatically when it receives a character via the serial port. So as long as the bit is not set the program repeats the "JNB" instruction continuously.

Once a character is received the RI bit will be set automatically and the above condition automatically fails and program flow falls through to the "MOV" instruction that reads the character into the Accumulator. The last instruction clears the RI bit to indicate that the last byte has been read.

Chapter 10: Interrupts

As the name implies, an interrupt is some event that interrupts normal program execution.

As stated earlier, program flow is always sequential, being altered only by those instructions that expressly cause program flow to deviate in some way. However, interrupts give us a mechanism to "put on hold" the normal program flow, execute a subroutine, and then resume normal program flow as if we had never left it. This subroutine, called an interrupt handler or interrupt service routine (ISR), is only executed when a certain event (interrupt) occurs. The event may be one of the timers "overflowing," receiving a character via the serial port, transmitting a character via the serial port, or one of two "external events." The 8052 may be configured so that when any of these events occur the main program is temporarily suspended and control passed to a special section of code, which presumably would execute some function related to the event that occurred. Once complete, control would be returned to the original program. The main program never even knows it was interrupted.

The ability to interrupt normal program execution when certain events occur makes it much easier and much more efficient to handle certain conditions. If it were not for interrupts we would have to manually check in our main program whether the timers had overflowed, whether we had received another character via the serial port, or if some external event had occurred. Besides making the main program ugly and hard to read, such a situation would make our program inefficient since we'd be burning precious instruction cycles checking for events that happen infrequently.

For example, let's say we have a large 16k program executing many subroutines performing many tasks. Let's also suppose that we want our program to automatically toggle the P3.0 port every time timer 0 overflows. The code to do this isn't too difficult:

```
        JNB TF0,SKIP_TOGGLE
        CPL P3.0
        CLR TF0
SKIP_TOGGLE: ...
```

Since the TF0 flag is set whenever timer 0 overflows, the above code will toggle P3.0 every time timer 0 overflows. This accomplishes what we want, but is inefficient.

The JNB instruction consumes 2 instruction cycles to determine that the flag is not set and jump over the unnecessary code. In the event that timer 0 overflows, the CPL and CLR instruction require 2 instruction cycles to execute. To make the math easy, let's say the rest of the code in the program requires 98 instruction cycles. In total, our code consumes 100 instruction cycles (98 instruction cycles plus the 2 that are executed every iteration to determine whether or not timer 0 has overflowed). If we're in 16-bit timer mode, timer 0 will overflow every 65,536 machine cycles. In that time we would have performed 655 JNB tests for a total of 1310 instruction cycles, plus another 2 instruction cycles to perform the code. So to achieve our goal we've spent 1312 instruction cycles. So 2.002% of our time is being spent just checking when to toggle P3.0. And our code is ugly because we have to make that check every iteration of our main program loop.

Luckily, this isn't necessary. Interrupts let us forget about checking for the condition. The microcontroller itself will check for the condition automatically and when the condition is met will jump to a subroutine (the interrupt handler), execute the code, then return. In this case, our subroutine would be nothing more than:

```
CPL P3.0      ;Toggle P3.0
RETI          ;Return from the interrupt
```

First, you'll notice the CLR TF0 command has disappeared. That's because when the 8052 executes our "timer 0 interrupt routine," it automatically clears the TF0 flag. You'll also notice that instead of a normal RET instruction we have a RETI instruction. The RETI instruction does the same thing as a RET instruction, but tells the 8051 that an interrupt routine has finished. You must always end your interrupt handlers with RETI.

Thus, every 65536 instruction cycles we execute the CPL instruction and the RETI instruction. Those two instructions together require 3 instruction cycles, and we've accomplished the same goal as the first example that required 1312 instruction cycles. As far as the toggling of P3.0 goes, our code is 437 times more efficient! Not to mention it's much easier to read and understand because we don't have to remember to always check for the timer 0 flag in our main program. We just setup the interrupt and forget about it, secure in the knowledge that the 8052 will execute our code whenever it's necessary.

The same idea applies to receiving data via the serial port. One way to do it is to continuously check the status of the RI flag in an endless loop. Or we could check the RI flag as part of a larger program loop. However, in the latter case we run the risk of missing characters--what happens if a character is received right after we do the check, the rest of our program executes, and before we even check RI a second character has come in. We will lose the first character. With interrupts, the 8052 will put the main program "on hold" and call our special routine to handle the reception of a character. Thus, we neither have to put an ugly check in our main code nor will we lose characters.

10.1 Events that can Trigger Interrupts

We can configure the 8052 so that any of the following events will cause an interrupt:

1. Timer 0 Overflow.
2. Timer 1 Overflow.
3. Reception/Transmission of Serial Character.
4. External Event 0.
5. External Event 1.

In other words, we can configure the 8052 so that when Timer 0 Overflows or when a character is sent/received the appropriate interrupt handler routines are called.

Obviously we need to be able to distinguish between various interrupts and be able to execute different code depending on what interrupt is triggered. This is accomplished by jumping to a fixed address when a given interrupt occurs.

Interrupt	Flag	Interrupt Handler Address
External 0	IE0	0003h
Timer 0	TF0	000Bh
External 1	IE1	0013h
Timer 1	TF1	001Bh
Serial	RI/TI	0023h

By consulting the above chart we see that whenever Timer 0 overflows (i.e., the TF0 bit is set), the main program will be temporarily suspended and control will jump to 000BH. It is assumed that we have code at address 000BH that handles the situation of Timer 0 overflowing.

10.2 Setting Up Interrupts

By default at power-up, all interrupts are disabled. This means that even if, for example, the TF0 bit is set, the 8052 will not execute the interrupt. Your program must specifically tell the 8052 that it wishes to enable interrupts and specifically which interrupts it wishes to enable.

Your program may enable and disable interrupts by modifying the IE SFR (A8h):

IE (A8h) SFR

Bit	Name	Bit Address	Explanation of Function
7	EA	AFh	Global Interrupt Enable/Disable
6	-	AEh	Undefined
5	-	ADh	Undefined
4	ES	ACh	Enable Serial Interrupt
3	ET1	ABh	Enable Timer 1 Interrupt
2	EX1	AAh	Enable External 1 Interrupt
1	ET0	A9h	Enable Timer 0 Interrupt
0	EX0	A8h	Enable External 0 Interrupt

As you can see, each of the 8052's interrupts has its own bit in the IE SFR. You enable a given interrupt by setting the corresponding bit. For example, if you wish to enable Timer 1 Interrupt, you would execute either:

```
MOV IE,#08h
```

or

```
SETB ET1
```

Both of the above instructions set bit 3 of IE, thus enabling Timer 1 Interrupt. Once Timer 1 Interrupt is enabled, whenever the TF1 bit is set, the 8052 will automatically put "on hold" the main program and execute the Timer 1 Interrupt Handler at address 001Bh.

However, before Timer 1 Interrupt (or any other interrupt) is truly enabled, you must also set bit 7 of IE. Bit 7, the Global Interrupt Enable/Disable, enables or disables all interrupts simultaneously. That is to say, if bit 7 is cleared then no interrupts will occur, even if all the other bits of IE are set. Setting bit 7 will enable all the interrupts that have been selected by setting other bits in IE. This is useful in program execution if you have time-critical code that needs to execute. In this case, you may need the code to execute from start to finish without any interrupt getting in the way. To accomplish this you can simply clear bit 7 of IE (**CLR EA**) and then set it after your time-critical code is done.

So, to sum up what has been stated in this section, to enable the Timer 1 Interrupt the most common approach is to execute the following two instructions:

```
SETB ET1      ;Enable Timer 1 Interrupt
SETB EA       ;Enable Global Interrupt flag
```

Thereafter, the Timer 1 Interrupt Handler at 01Bh will automatically be called whenever the TF1 bit is set (upon Timer 1 overflow).

10.3 Polling Sequence

The 8052 automatically evaluates whether an interrupt should occur after every instruction. When checking for interrupt conditions, it checks them in the following order:

1. External 0 Interrupt
2. Timer 0 Interrupt
3. External 1 Interrupt
4. Timer 1 Interrupt
5. Serial Interrupt

This means that if a Serial Interrupt occurs at the exact same instant that an External 0 Interrupt occurs, the External 0 Interrupt will be executed first and the Serial Interrupt will be executed once the External 0 Interrupt has completed.

10.4 Interrupt Priorities

The 8052 offers two levels of interrupt priority: high and low. By using interrupt priorities you may assign higher priority to certain interrupt conditions.

For example, you may have enabled timer 1 interrupt that is automatically called every time timer 1 overflows. Additionally, you may have enabled the serial interrupt that is called every time a character is received via the serial port. However, you may consider that receiving a character is much more important than the timer interrupt. In this case, if timer 1 interrupt is already executing you may wish that the serial interrupt itself interrupts the timer 1 interrupt. When the serial interrupt is complete, control passes back to timer 1 interrupt and finally back to the main program. You may accomplish this by assigning a high priority to the Serial Interrupt and a low priority to the Timer 1 Interrupt.

Interrupt priorities are controlled by the IP SFR (B8h). The IP SFR has the following format:

IP (B8h) SFR

Bit	Name	Bit Address	Explanation of Function
7	-	BFh	Undefined
6	-	Beh	Undefined
5	-	BDh	Undefined
4	PS	BCh	Serial Interrupt Priority
3	PT1	BBh	Timer 1 Interrupt Priority
2	PX1	Bah	External 1 Interrupt Priority
1	PT0	B9h	Timer 0 Interrupt Priority
0	PX0	B8h	External 0 Interrupt Priority

When considering interrupt priorities, the following rules apply:

1. Nothing can interrupt a high-priority interrupt--not even another high priority interrupt.
2. A high-priority interrupt may interrupt a low-priority interrupt.
3. A low-priority interrupt may only occur if no other interrupt is already executing.
4. If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both interrupts are of the same priority the interrupt that is serviced first by polling sequence will be executed first.

10.5 Interrupt Triggering

When an interrupt is triggered, the following actions are taken automatically by the microcontroller:

1. The current Program Counter is saved on the stack, low-byte first, high-byte second.
2. Interrupts of the same and lower priority are blocked.
3. In the case of Timer and External interrupts, the corresponding interrupt flag is cleared.
4. Program execution transfers to the corresponding interrupt handler vector address.
5. The Interrupt Handler routine, written by the developer, is executed.

Take special note of the third step: If the interrupt being handled is a Timer or External interrupt, the microcontroller automatically clears the interrupt flag before passing control to your interrupt handler routine. This means it is not necessary that you clear the bit in your code.

10.6 Exiting Interrupt

An interrupt ends when your program executes the **RETI** (Return from Interrupt) instruction. When the RETI instruction is executed the following actions are taken by the microcontroller:

1. Two bytes are popped off the stack into the Program Counter to restore normal program execution.
2. Interrupt status is restored to its pre-interrupt status.

10.7 Serial Interrupts

Serial Interrupts are slightly different than the rest of the interrupts. This is due to the fact that there are two interrupt flags: RI and TI. If either flag is set, a serial interrupt is triggered. As you will recall from the section on the serial port, the RI bit is set when a byte is received by the serial port and the TI bit is set when a byte has been sent.

This means that when your serial interrupt is executed it may have been triggered because the RI flag was set or because the TI flag was set--or because both flags were set. Thus, your routine must check the status of these flags to determine what action is appropriate. Also, since the 8052 does not automatically clear the RI and TI flags you must clear these bits in your interrupt handler.

A brief code example is in order:

```
INT_SERIAL:
       JNB RI,CHECK_TI     ;If RI flag is not set, we jump to check TI
       MOV A,SBUF          ;If we got here, the RI bit *was* set
       CLR RI              ;Clear the RI bit after we've processed it
CHECK_TI:
       JNB TI,EXIT_INT     ;If TI flag not set, we jump to the exit point
       CLR TI              ;Clear TI bit before we send next character
       MOV SBUF,#'A'             ;Send another character to the serial
port
EXIT_INT:
       RETI                ;Exit interrupt handler
```

As you can see, our code checks the status of both interrupts flags. If both flags were set, both sections of code will be executed. Also note that each section of code clears its corresponding interrupt flag. If you

forget to clear the interrupt bits, the serial interrupt will be executed over and over until you clear the bit. For this reason it is very important that you always clear the interrupt flags in a serial interrupt.

10.8 Register Protection

One very important rule applies to all interrupt handlers: interrupts must leave the processor in the same state as it was in when the interrupt initiated. Remember, the idea behind interrupts is that the main program isn't aware that they are executing in the "background." However, consider the following code:

```
CLR C         ;Clear carry
MOV A,#25h    ;Load the accumulator with 25h
ADDC A,#10h   ;Add 10h, with carry
```

After the above three instructions are executed, the accumulator will contain a value of 35h.

But what would happen if an interrupt occurred right after the MOV instruction. During this interrupt, the carry bit was set and the value of the accumulator was changed to 40h. When the interrupt finished and control was passed back to the main program, the ADDC would add 10h to 40h, and additionally add an additional 01h because the carry bit is set. In this case, the accumulator will contain the value 51h at the end of execution.

In this case, the main program has seemingly calculated the wrong answer. How can 25h + 10h yield 51h as a result? It doesn't make sense. A developer that was unfamiliar with interrupts would be convinced that the microcontroller was damaged in some way, provoking problems with mathematical calculations.

What has happened, in reality, is the interrupt did not protect the registers it used. Restated: *An interrupt must leave the processor in the same state as it was in when the interrupt initiated.*

This means if your interrupt uses the accumulator, it must insure that the value of the accumulator is the same at the end of the interrupt as it was at the beginning. This is generally accomplished with a PUSH and POP sequence at the beginning and end of each interrupt handler. For example:

```
INTERRUPT_HANDLER:
       PUSH ACC      ;Push the initial value of accumulator onto stack
       PUSH PSW      ;Push the initial value of PSW SFR onto stack
       MOV A,#0FFh   ;Use accumulator & PSW for whatever you want
       ADD A,#02h    ;Use accumulator & PSW for whatever you want
       POP PSW       ;Restore the initial value of the PSW from the stack
       POP ACC       ;Restore initial value of the accumulator from stack
```

The guts of the interrupt are the MOV instruction and the ADD instruction. However, these two instructions modify the Accumulator (the MOV instruction) and also modify the value of the carry bit (the ADD instruction will cause the carry bit to be set). Since an interrupt routine must guarantee that the registers remain unchanged by the routine, the routine pushes the original values onto the stack using the PUSH instruction. It is then free to use the registers it protected to its heart's content. Once the interrupt has finished its task, it POPs the original values back into the registers. When the interrupt exits, the main program will never know the difference because the registers are exactly the same as they were before the interrupt executed.

In general, your interrupt routine must protect the following registers if they are used within your interrupt routine:

1. Program Status Word SFR (PSW)
2. Data Pointer SFRs (DPH/DPL)
3. Accumulator (ACC)
4. “B” Register (B)
5. “R” Registers (R0-R7)

Remember that PSW consists of many individual bits that are set by various 8052 instructions. Unless you are absolutely sure of what you are doing and have a complete understanding of what instructions set what bits, it is generally a good idea to always protect PSW by pushing and popping it off the stack at the beginning and end of your interrupts.

Note also that most assemblers (in fact, ALL assemblers that I know of) will not allow you to execute the instruction:

```
PUSH R0       ;Error - Invalid instruction!
```

This is due to the fact that, depending on which register bank is selected, R0 may refer to either internal RAM address 00h, 08h, 10h, or 18h. R0, in and of itself, is not a valid memory address that the PUSH and POP instructions can use. Thus, if you are using any "R" register in your interrupt routine, you will have to do an additional step by first moving the "R" register to the Accumulator and then pushing the Accumulator on the stack. For example, instead of PUSH R0 you would execute:

```
MOV A,R0      ;Copies the contents of R0 to the Accumulator
PUSH ACC      ;Pushes the Accumulator (which holds R0) onto the stack
```

When your interrupt is done, you'd restore the "R" register with code such as:

```
POP ACC       ;Pops the last value on stack into Accumulator
MOV R0,A      ;Copies the contents of the Accumulator into R0
```

Of course, you must first protect the accumulator itself by pushing the Accumulator onto the stack *before* you start using it to push "R" registers onto the stack and when you are done restoring the "R" registers at the end of your routine you must restore the Accumulator itself with one additional POP ACC instruction.

10.9 Common Problems with Interrupts

Interrupts are a very powerful tool available to the 8052 developer but, when used incorrectly, can be a source of a huge number of debugging hours. Errors in interrupt routines are often very difficult to diagnose and correct.

If you are using interrupts and your program is crashing or does not seem to be performing as you would expect, always review the following interrupt-related issues:

1. **Register Protection**: Make sure you are protecting all your registers, as explained above. If you forget to protect a register that your interrupt is using, very strange results may occur. In our example above we saw how failure to protect registers caused the main program to apparently calculate that 25h + 10h = 51h. If you witness problems with registers changing

values unexpectedly or operations producing "incorrect" values, it is very likely that you've forgotten to protect registers. ALWAYS PROTECT YOUR REGISTERS.

2. **Forgetting to restore protected values**: Another common error is to push registers onto the stack to protect them, and then forget to pop them off the stack before exiting the interrupt. For example, you may push ACC, B, and PSW onto the stack in order to protect them and subsequently pop only ACC and PSW off the stack before exiting. In this case, since you forgot to restore the value of "B", an extra value remains on the stack. When you execute the RETI instruction the 8051 will use that value as the return address instead of the correct value. In this case, your program will almost certainly crash. ALWAYS MAKE SURE YOU POP THE SAME NUMBER OF VALUES OFF THE STACK AS YOU PUSHED ONTO IT.

3. **Using RET instead of RETI**: Remember that interrupts are always terminated with the RETI instruction. It is easy to inadvertently use the RET instruction instead. However, the RET instruction will not end your interrupt. Usually, using a RET instead of a RETI will cause the illusion of your main program running normally, but your interrupt will only be executed once. If it appears that your interrupt mysteriously stops executing, verify that you are exiting with RETI.

Simulators, such as Vault Information Services' 8052 Simulator for Windows or Pinnacle 52 (http://www.vaultbbs.com/sim8052, http://www.vaultbbs.com/pinnacle), contain special features that will notify you if you fail to protect registers or commit other common interrupt-related errors.

Chapter 11: 8052 Assembly Language

Assembly language is a low-level, pseudo-English representation of the microcontroller's machine language. Each assembly language instruction has a one-to-one relation to one of the microcontroller's machine-level instructions.

High-level languages, such as 'C', Basic, Visual Basic, etc. are one or more steps above assembly language in that no significant knowledge of the underlying architecture is necessary. It is possible (and common) for a developer to program a Visual Basic application in Windows without knowing much of anything about the Windows API, much less the underlying architecture of the Intel Pentium. Further, a developer who has written code in 'C' for Unix won't have significant problems adapting to writing code in 'C' for Windows or a microcontroller such as an 8052—while there are some variations, the C compiler itself takes care of most of the processor-specific issues.

Assembly language, on the other hand, is *very* processor-specific. While a prior knowledge of assembly language with any given processor will be helpful when attempting to begin coding in the assembly language of another processor, the two assembly languages may be extremely different. Different architectures have different instruction sets and different forms of addressing. In fact, only general concepts may "port" from one processor to another.

The low-level nature of assembly language programming requires an understanding of the underlying architecture of the processor for which one is developing. This is why we explained the 8052's architecture fully before attempting to introduce the reader to assembly language programming in this document. Many aspects of assembly language may be completely confusing without a prior knowledge of the architecture.

This section of the document will introduce the reader to 8052 assembly language, concepts, and programming style.

11.1 Syntax

Each line of an assembly language program consists of the following syntax, each field of which is optional. However, when used, the elements of the line must appear in the following order:

- **Label** – A user-assigned symbol that defines this instruction's address in memory. The label, if present, must be terminated with a colon.
- **Instruction** – An assembly language instruction which, when assembled, will perform some specific function when executed by the microcontroller. The instruction is a psuedo-English "mnemonic" which relates directly to one machine language instruction.
- **Comment** – The developer may include a comment on each line for in-line documentation. These comments are ignored by the assembler but may make it easier to subsequently understand the code. A comment, if used, must be preceded with a semicolon.

In summary, then, a typical 8052 assembly language line might appear as:

```
MYLABEL: MOV A,#25h ;This is just a sample comment
```

In this line, the label is **MYLABEL**. This means that if subsequent instructions in the program need to make reference to this instruction, they may do so by referring to the label "MYLABEL" rather than the memory address of the instruction.

The 8052 assembly language instruction in this line is **MOV A,#25h**. This is the actual instruction that the assembler will analyze and assemble into the two bytes **74h 25h**. The first number, 74h, is the 8052 machine language instruction (opcode) "MOV A,#*dataValue*"—which means "Move the value *dataValue* into the accumulator." In this case, the value of *dataValue* will be the value of the byte that immediately follows the opcode. Since we want to load the accumulator with the value 25h, the byte following the opcode is 25h. As you can see, there is a one-to-one relationship between the assembly language instruction and the machine language code that is generated by the assembler.

Finally, the instruction above includes the optional comment **;This is just a sample comment**. The comment must always start with a semi-colon. The semi-colon tells the assembler that the rest of the line is a comment that should be ignored by the assembler.

Since all fields are optional, the following are also alternatives to the above syntax:

Label without instruction and comment:	`LABEL:`
Line with label and instruction:	`LABEL: MOV A,#25h`
Line with instruction and comment:	`MOV A,#25h ;This is a comment`
Line with label and a comment:	`LABEL: ;This is a comment`
Line with just a comment:	`;This is a comment`

All of the above permutations are completely valid. It is up to the developer which components of the assembly language syntax will be used. But, when used, they must follow the above syntax and in the correct order.

NOTE: It does not matter what column each field begins in. That is, a label may start at the beginning of the line or after any number of blank spaces. Likewise an instruction may start in any column of the line, as long as it follows any label that may also be on that line.

11.2 Number Bases

Most assemblers are capable of accepting numeric data in a variety of number bases. Commonly supported are decimal, hexadecimal, binary, and octal.

Decimal: To express a decimal number in assembly language, simply enter the number as normal.

Hexadecimal: To express a hexadecimal number, enter the number as a hexadecimal value, and terminate the number with the suffix "h". For example, the hexadecimal number 45 would be expressed as 45h. Further, if the hexadecimal number begins with an alphabetic character (A, B, C, D, E, or F), the number must be preceded with a leading zero. For example, the hex number E4 would be written as 0E4h. The leading zero allows the assembler to differentiate the hex number from a symbol since a symbol can never start with a number.

Binary: To express a binary number, enter the binary number itself followed by a trailing "B", to indicate binary. For example, the binary number 100010 would be expressed as 100010B.

Octal: To express an octal number, enter the octal number itself followed by a trailing "Q", to indicate octal. For example, the octal number 177 would be expressed as 177Q.

For example, all of the following instructions load the accumulator with 30 (decimal):

```
MOV A,#30
MOV A,#11110B
MOV A,#1EH
MOV A,#36Q
```

11.3 Expressions

You may use mathematical expressions in your assembly language instructions anywhere where a numeric value may be used. For example, both of the following are valid assembly language instructions:

```
MOV A,#20h + 34h          ;Equivalent to #54h
MOV 35h + 2h,#10101B      ;Equivalent to MOV 37h,#10101B
```

11.4 Operator Precedence

Mathematical operators within an expression are subject to the following order of precedence. Operators at the same "level" are evaluated left-to-right.

1 (Highest)	( )
2	HIGH LOW
3	* / MOD SHL SHR
4	EQ NE LT LE GT GE = <> < <= > >=
5	NOT
6	AND
7 (Lowest)	OR XOR

NOTE: If you have any doubt about operator precedence, it is useful to use parentheses to force the order of evaluation that you have contemplated. Even if it isn't strictly necessary, it is often easier to read mathematical expressions when parentheses have been added, even when the parentheses aren't technically necessary.

11.5 Characters and Character Strings

Characters and character strings are enclosed in single-quotes and are converted to their numeric equivalent at assemble-time. For example, the following two instructions are the same:

```
MOV A,#'C'
MOV A,#43H
```

The two instructions are the same because the assembler will see the 'C' sequence, convert the character contained in quotes to its ASCII equivalent (43H), and use that value. Thus the second instruction is the same as the first.

Strings of characters are sometimes enclosed in single-quotes and sometimes enclosed in double-quotes. For example, Pinnacle 52 uses double-quotes to indicate a string of characters and a single-quote to indicate a single character. Thus:

```
MOV A,#'C'       ;Single character - ok
MOV A,#"STRING" ;String - ERROR! Can't load a string into accumulator
```

Strings are invalid in the above context, although there are other special assembler directives that do allow strings. Be sure to check the manual for your assembler to determine whether character strings should be placed within single-quotes or double-quotes.

11.6 Changing Program Flow (LJMP, SJMP, AJMP)

LJMP, SJMP and **AJMP** are used as a "go to" in assembly language. They cause program execution to continue at the address or label they specify. For example:

```
LJMP LABEL3          ;Program execution is transferred to LABEL3
LJMP 2400h           ;Program execution is transferred to address 2400h
SJMP LABEL4          ;Program execution is transferred to LABEL4
AJMP LABEL7          ;Program execution is transferred to LABEL7
```

The differences between LJMP, SJMP, and AJMP are:

1. **LJMP** requires 3 bytes of program memory and can jump to any address in the program.
2. **SJMP** requires 2 bytes of program memory, but can only jump to an address within 128 bytes of itself.
3. **AJMP** requires 2 bytes of program memory, but can only jump to an address in the same 2k block of memory.

These instructions perform the same task, but differ in what addresses they can jump to and how many bytes of program memory they require.

LJMP will always work. You can always use LJMP to jump to any address in your program.

SJMP requires two bytes of memory but has the restriction that it can only jump to an instruction or label that is within 128 bytes before or 127 bytes after the instruction. This is useful if you are branching to an address that is very close to the jump itself. You save 1 byte of memory by using SJMP instead of AJMP.

AJMP also requires two bytes of memory but has the restriction that it can only jump to an instruction or label that is in the same 2k block of program memory. For example, if the AJMP instruction is at address 0200h it could only jump to addresses between 0000h and 07FFh. It could not jump to 800h, for example.

NOTE: Some optimizing assemblers allow you to use JMP in your code. While there isn't a JMP instruction in the 8052 instruction set, the optimizing assembler will automatically replace your JMP with the most memory-efficient instruction. That is, it will try to use SJMP or AJMP if it is possible, but will resort to LJMP if necessary. This allows you to simply use the JMP instruction and let the assembler worry about saving program memory whenever it is possible.

11.7 Subroutines (LCALL, ACALL, RET)

As in other languages, 8052 assembly language permits the use of subroutines. A subroutine is a section of code that is called by a program, does a task, and then returns to the instruction immediately following that of the instruction that made the call.

LCALL and **ACALL** are both used to call a subroutine. LCALL requires three bytes of program memory and can call any subroutine anywhere in memory; ACALL requires two bytes of program memory and can only call a subroutine within the same 2k block of program memory.

Both call instructions will save the current address on the stack and jump to the specified address or label. The subroutine at that address will perform whatever task it needs to and then return to the original instruction by executing the **RET** instruction.

For example, consider the following code:

```
             LCALL SUBROUTINE1   ;Call the SUBROUTINE1 subroutine
             LCALL SUBROUTINE2   ;Call the SUBROUTINE2 subroutine
             .
             .
             .
SUBROUTINE1: {subroutine code}   ;Insert subroutine code here
             RET                 ;Return from subroutine

SUBROUTINE2: {subroutine code}   ;Insert subroutine code here
             RET                 ;Return from subroutine
```

The code above starts by calling SUBROUTINE1. Execution will transfer to SUBROUTINE1 and execute whatever code is found there. When the MCU hits the RET instruction it will automatically return to the next instruction, which is LCALL SUBROUTINE2. SUBROUTINE2 will then be called, execute its code, and return to the main program when it hits its RET instruction.

NOTE: It is very important that you end all subroutines with the RET instruction, and that all subroutines exit themselves by executing the RET instruction. If you call a subroutine with LCALL or ACALL and don't execute a corresponding RET. Failing to do so will lead to unpredictable results.

NOTE: Subroutines may call other subroutines. For example, in the code above SUBROUTINE1 could include an instruction that calls SUBROUTINE2. SUBROUTINE2 would then execute and return to SUBROUTINE1 which would then return to the instruction that called it. However, keep in mind that every LCALL or ACALL you execute expands the stack by two bytes. If your stack starts at Internal RAM address 30h and you make 10 successive calls to subroutines from within subroutines your stack will expand by 20 bytes to 44h.

NOTE: Recursive subroutines (subroutines that call themselves) are a very popular method of solving some common programming problems. However, unless you know for certain that the subroutine will call itself a certain number of times, it is generally not possible to use subroutine recursion in 8052 assembly language. Due to the small amount of Internal RAM a recursive subroutine could quickly cause the stack to fill all of Internal RAM.

11.8 Register Assignment (MOV)

One of the most commonly used 8052 assembly language instructions, and the first we will introduce here, is the **MOV** instruction. 57 of the 254 opcodes are MOV instructions which is due to the fact that there are many ways you can move data between various registers using various addressing modes.

The MOV instruction is used to "move" data from one register to another—or to simply assign a value to a register—and has the following general syntax:

MOV *DestinationRegister,SourceValue*

"*DestinationRegister*" always indicates the register or address in which "*SourceValue*" will be stored, whereas *SourceValue* indicates the register the value will be taken from, or the value itself if it is preceded by a pound sign (#).

For example:

```
MOV A,25h     ;Moves contents of Internal RAM address 25h to accumulator
MOV 25h,A     ;Move contents of accumulator into Internal RAM address 25h
MOV 80h,A     ;Move the contents of the accumulator to P0 SFR (80h)
MOV A,#25h    ;Moves the value 25h into the accumulator
```

As you can see, the first parameter is the register, Internal RAM address, or SFR address that a value is being moved to. Another way of looking at it is that the first parameter is the register that is going to be assigned a new value.

Likewise, the second parameter tells the 8052 where to get the new value. Normally, the value of the second parameter indicates the Internal RAM or SFR address from which the value should be obtained. However, if the second parameter is preceded by a pound sign (#), the register will be assigned the *value* of the number that follows the pound sign (as is demonstrated in the last example above).

As already mentioned, the MOV instruction is one of the most common and vital instructions that an 8052 assembly language programmer will use. The prospective assembly language programmer must fully master the MOV instruction. This may seem simple but it requires knowing all of the permutations of the MOV instruction and knowing when to use them. This knowledge comes with time and experience, and by reviewing the "8052 Instruction Set Overview" (Appendix A).

It is important that all types of MOV instructions be understood so that the programmer knows what types of MOV instructions are available as well as what kinds of MOV instructions are *not* available.

Careful inspection of the MOV commands in the instruction set reference will reveal that there is no "MOV from 'R' register to 'R' register." That is to say, the following instruction is invalid:

```
MOV R2,R1     ;INVALID!!
```

This is a logical type of operation for a programmer to want to implement, but the instruction above is not valid because there is no "MOV from 'R' register to 'R' register" MOV instruction. Instead, the above must be programmed as:

```
MOV A,R1      ;Move R1 to accumulator
MOV R2,A      ;Move accumulator to R2
```

Another combination that is not supported is "MOV indirectly from Internal RAM to another Indirect RAM address". Again, the following instruction is invalid:

```
MOV @R0,@R1   ;INVALID!!
```

This is not a valid MOV combination. Instead, this could be programmed as:

```
MOV A,@R1     ;Move contents of IRAM pointed to by R1 to accumulator
MOV @R0,A     ;Move accumulator to Internal RAM address pointed to by R0
```

Also note that only R0 and R1 can be used for "Indirect Addressing".

Programming Hint: When you find yourself in a situation that you need to execute a type of MOV instruction that doesn't exist, it is generally helpful to use the accumulator. If a given MOV instruction doesn't exist it can usually be accomplished by using two MOV instructions that both use the accumulator as a transfer or temporary register.

With this knowledge of the MOV instruction, we can perform some simple memory assignment tasks.

1. **Clear the contents of Internal RAM address FFh:**

```
MOV A,#00h    ;Move the value 00h to the accumulator (Accumulator=00h)
MOV R0,#0FFh  ;Move the value FFh to R0 (R0=0FFh)
MOV @R0,A     ;Move accumulator to @R0, thus clearing contents of FFh
```

2. Clear the contents of Internal RAM address FFh (more efficient):

```
MOV R0,#0FFh  ;Move the value FFh to R0 (R0=0FFh)
MOV @R0,#00h  ;Move 00h to @R0 (FFh), thus clearing contents of FFh
```

3. Clear the contents of all bit memory (Internal RAM addresses 20h through 2Fh):
(Note that this example will later be improved upon to require less code)

```
MOV 20h,#00h  ;Clear Internal RAM address 20h
MOV 21h,#00h  ;Clear Internal RAM address 21h
MOV 22h,#00h  ;Clear Internal RAM address 22h
MOV 23h,#00h  ;Clear Internal RAM address 23h
MOV 24h,#00h  ;Clear Internal RAM address 24h
MOV 25h,#00h  ;Clear Internal RAM address 25h
MOV 26h,#00h  ;Clear Internal RAM address 26h
MOV 27h,#00h  ;Clear Internal RAM address 27h
MOV 28h,#00h  ;Clear Internal RAM address 28h
MOV 29h,#00h  ;Clear Internal RAM address 29h
MOV 2Ah,#00h  ;Clear Internal RAM address 2Ah
MOV 2Bh,#00h  ;Clear Internal RAM address 2Bh
MOV 2Ch,#00h  ;Clear Internal RAM address 2Ch
MOV 2Dh,#00h  ;Clear Internal RAM address 2Dh
MOV 2Eh,#00h  ;Clear Internal RAM address 2Eh
MOV 2Fh,#00h  ;Clear Internal RAM address 2Fh
```

11.9 Incrementing and Decrementing Registers (INC/ DEC)

Two instructions, **INC** and **DEC**, can be used to increment or decrement the value of a register, Internal RAM, or SFR by 1. These instructions are rather self-explanatory.

The INC instruction will add 1 to the current value of the specified register. If the current value is 255, it will "overflow" back to 0. For example, if the accumulator holds the value 240 and the INC A instruction is executed, the accumulator will be incremented to 241.

```
INC A         ;Increment the accumulator by 1
INC R1        ;Increment R1 by 1
INC 40h       ;Increment Internal RAM address 40h by 1
```

The DEC instruction will subtract 1 from the current value of the specified register. If the current value is 0, it will "underflow" back to 255. For example, if the accumulator holds the value 240 and the DEC A instruction is executed, the accumulator will be decremented to 239.

```
DEC A         ;Decrement the accumulator by 1
DEC R1        ;Decrement R1 by 1
DEC 40h       ;Decrement Internal RAM address 40h by 1
```

NOTE: If you've programmed in assembly language under other architectures, you may be used to the INC and DEC instructions setting an overflow or underflow flag when the register overflows from 255 to 0 or underflows from 0 back to 255. This is not the case with the INC and DEC instructions in 8052 assembly language. Neither of these instructions affects any flags whatsoever.

11.10 Program Loops (DJNZ)

Many operations are conducted within finite loops. That is, a given code segment is executed repeatedly until a given condition is met.

A common type of loop is a simple "counter loop." That is, you execute the code segment a certain number of times and finish. This is accomplished easily in 8052 assembly language with the **DJNZ** instruction. DJNZ means "Decrement, Jump if Not Zero". Consider the following code:

```
       MOV R0,#08h   ;Set number of loop cycles to 8
LOOP:  INC A         ;Increment acc (or do whatever the loop does)
       DJNZ R0,LOOP  ;Decrement R0, loop back to LOOP if R0 is not 0
       DEC A         ;Decrement accumulator (or whatever you want to do)
```

This is a very simple counter loop. The first line initializes R0 to 8, which will be the number of times the loop will be executed. The second line, "LOOP", is the actual body of the loop. This could contain any instruction or instructions you wish to execute repeatedly. In this case, we just increment the accumulator with the INC A instruction.

The interesting part is the last line with the DJNZ instruction. This instruction reads "Decrement the R0 Register, and if it's not now zero jump back to LOOP". This instruction will decrement the R0 register. It will then check to see if the new value is zero and, if it isn't, will go back to LOOP. The first time this loop executes R0 will be decremented from 08 to 07, then 07 to 06, and so on until it decrements from 01 to 00. At that point, the DJNZ instruction will fail since the accumulator is *zero*; this will cause the program to *not* go back to LOOP and, thus, it will continue executing with the DEC instruction—or whatever you want your program to do after the loop is complete.

DJNZ is one of the most common ways to perform programming loops that execute a specific number of times. The number of times the loop will be executed depends on the initial value of the "R" register that is used by the DJNZ instruction.

11.11 Setting, Clearing, and Moving Bits (SETB/CLR/CPL/MOV)

One very powerful feature of the 8052 architecture is its ability to manipulate individual bits on a bit-by-bit basis. As mentioned earlier in this document, there are 128 numbered bits (00h through 7Fh) that may be used by the user's program as bit variables. Additionally, bits 80h through FFh allow access to SFRs that are divisible by 8 on a bit-by-bit basis. The two basic instructions to manipulate bits are **SETB** and **CLR** while a third instruction, **CPL**, is also often used.

The SETB instruction will set the specified bit, which means the bit will then have a value of "1", or "on". For example:

```
SETB 20h      ;Sets user bit 20h (sets bit 0 of IRAM address 24h to 1)
SETB 80h      ;Sets bit 0 of SFR 80h  (P0) to 1
SETB P0.0     ;Exactly the same as the previous instruction
SETB C        ;Sets the carry bit to 1
SETB TR1      ;Sets the TR1 bit to 1 (turns on timer 1)
```

As illustrated by these instructions, SETB can be used in a variety of circumstances.

The first example, **SETB 20h**, sets user bit 20h. Since all bits between 00h and 7Fh are user bits, you know that this corresponds to a user-defined bit. Since these 128 user bits reside in Internal RAM at the addresses of 20h through 2Fh, it is clear that bit 20h is the 32nd user-defined bit. Each byte of Internal RAM, by definition, holds 8 individual bits so bit 20h would be the lowest bit of Internal RAM 24h.

NOTE: It is very important that you understand that Bit Memory is a part of Internal RAM. In the case of SETB 20h we concluded that bit 20h is actually the low bit of Internal RAM address 24h. This is because bits 00h-07h are Internal RAM address 20h, bits 08h-0Fh are Internal RAM address 21h, bits 10h-17h are Internal RAM address 22h, bits 18h-1Fh are Internal RAM address 23h, and bits 20h-27h are Internal RAM address 24h.

The second example, **SETB 80h**, is similar to SETB 20h. Of course, SETB 80h sets bit 80h. However, remember that bits 80h-FFh correspond to individual bits of SFRs, not Internal RAM. Thus SETB 80h will actually set bit 0 of SFR 80h which is the P0 SFR.

The next instruction, **SETB P0.0**, is identical to SETB 80h. The only difference is that we are now referencing the bit by name rather than number. This will make your assembly language code more readable. The assembler will automatically convert "P0.0" to 80h when you assemble your program.

The next example, **SETB C**, is a slightly special case. This instruction sets the carry bit, which is a very important bit that is used for many purposes. It is also special in that there is an opcode that means "SETB C". While other SETB instructions require two bytes of program memory, the SETB C instruction only requires one.

Finally, the **SETB TR1** example shows a typical use of SETB to set an individual bit of an SFR. In this case, TR1 is TCON.6 (bit 6 of TCON SFR, SFR address 88h). Since TCON's SFR address is 88h it is divisible by 8 and, thus, addressable on a bit-by-bit basis.

The CLR instruction functions in the same manner, but clears the specified bit. For example:

```
CLR 20h      ;Clears user bit 20h to 0
CLR P0.0     ;Sets bit 0 of P0 to 0
CLR TR1      ;Clears TR1 bit to 0 (stops timer 1)
```

These two instructions, CLR and SETB, are the two fundamental instructions used to manipulate individual bits in 8052 assembly language.

A third bit instruction, **CPL**, complements the value of the given bit. The instruction syntax is exactly the same as SETB and CLR, but CPL will "flip the bit." If the bit was cleared, CPL will set it; likewise, if the bit was set it will be cleared.

NOTE: An additional instruction, **CLR A**, exists which is used to clear the contents of the accumulator. This is the only CLR instruction that clears an entire SFR rather than just a single bit. The CLR A instruction is the equivalent of MOV A,#00h. The advantage of using CLR A is that it requires only one byte of program memory whereas the MOV A,#00h solution requires two bytes. An additional instruction, **CPL A**, also exists. This instruction will flip (complement) each bit in the Accumulator. Thus if the Accumulator holds 255 (11111111 binary), it will hold 0 (00000000 binary) after the CPL A instruction has executed.

Finally, the MOV instruction can be used to move bit values between any given bit—user or SFR bits—and the Carry bit. The instructions **MOV C,*bit*** and **MOV *bit*,C** allow these bit movements to occur. They function like the MOV instruction described earlier, moving the value of the second bit to the value of the first bit.

Consider the following examples:

```
MOV C,P0.0   ;Move the value of the P0.0 line to the Carry bit
MOV C,30h    :Move the value of user bit 30h to the Carry bit
MOV 25h,C    ;Move the Carry bit to user bit 25h
```

These combination of MOV instructions that allow bits to be moved through the Carry flag allow for more advanced bit operations without the need for "workarounds" that would be required to move bit values if it weren't for these MOV instructions.

NOTE: The MOV instruction, when used with bits, can only move bit values to the Carry bit and from the Carry bit. There is no instruction that allows you to copy directly from one bit to the other bit with neither bit being the Carry bit. Thus it is often necessary to use the Carry bit as a temporary bit register to move a bit value from one user bit to another user bit.

11.12 Bit-Based Decisions & Branching (JB, JBC, JNB, JC, JNC)

It is often useful, especially in microcontroller applications, to execute different code based on whether or not a given bit is set or cleared. The 8052 instruction set offers five instructions that do precisely that.

JB means "Jump if bit set". The MCU will check the specified bit and, if it is set, will jump to the specified address or label.

JBC means "Jump if bit set, and clear Bit". This instruction is identical to JB except that if the bit will be cleared if it was set. That is to say, if the specified bit is set, the MCU will jump to the specified address or label and also clear the bit. In some cases this can save the programmer the use of an extra CLR instruction.

JNB means "Jump if bit not set". This instruction is the opposite of JB—it tests the specified bit and will jump to the specified label or address if the bit was not set.

JC means "Jump if carry set." This is the same as the JB instruction but it only tests the carry bit. Since many operations and decisions are based on whether or not the carry flag is set, an additional instruction was included in the instruction set to test for this common condition. Thus insead of using the instruction "JB C,label", which takes 3 bytes of program memory, the programmer may use "JC label" which only takes 2.

JNC means "Jump if carry bit not set." This is the opposite of JC. This instruction tests the carry bit and will jump to the specified label or address if the carry bit is clear.

Some examples of these instructions are:

```
JB 40h,LABEL1       ;Jumps to LABEL1 if user bit 40h is set
JBC 45h,LABEL2      ;Jumps to LABEL2 if user bit 45h set, then clears it
JNB 50h,LABEL3      ;Jumps to LABEL3 if user bit 50h is clear
JC LABEL4           ;Jumps to LABEL4 if the carry bit is set
JNC LABEL5          ;Jumps to LABEL5 if the carry bit is clear
```

These instructions are very common, and very useful. Virtually all 8052 assembly language programs of any complexity will use them—especially the JC and JNC instructions.

11.13 Value Comparison (CJNE)

CJNE (Compare, Jump if Not Equal) is a very important instruction. It is used to compare the value of a register to another value and branch to another instruction based on whether or not the values are the same. This is a very common way of building a switch…case decision structure or an IF…THEN…ELSE structure in assembly language.

The CJNE instruction compares the values of the first two parameters of the instruction and jumps to the address contained in the third parameter if the first two parameters are *not* equal.

```
CJNE A,#24h,NOT24   ;Jumps to the label NOT24 if accumulator isn't 24h
CJNE A,40h,NOT40    ;Jumps to the label NOT40 if accumulator is
                    ;different than the value contained in Internal RAM
                    ;address 40h
```

```
CJNE R2,#36h,NOT36  ;Jumps to the label NOT36 if R2 isn't 36h
CJNE @R1,#25h,NOT25 ;Jumps to the label NOT25 if the Internal RAM
                    ;address pointed to by R1 does not contain 25h
```

As illustrated above, the MCU will compare the first parameter to the second parameter. If they are different then it will jump to the label provided; if the two values are the same then execution will continue with the next instruction. This can allow the programming of extensive condition evaluations.

For example, we may want to call the PROC_A subroutine if the Accumulator is equal to 30h, call the CHECK_LCD subroutine if the Accumulator equals 42h, and call the DEBOUNCE_KEY subroutine if the accumulator equals 50h. This could be implemented using CJNE as follows:

```
             CJNE A,#30h,CHECK2  ;If A is not 30h, jump to CHECK2 label
             LCALL PROC_A        ;If A is 30h, call the PROC_A subroutine
             SJMP CONTINUE       ;When we get back, we jump to CONTINUE label
CHECK2:      CJNE A,#42h,CHECK3  ;If A is not 42h, jump to CHECK3 label
             LCALL CHECK_LCD     ;If A is 42h, call the CHECK_LCD subroutine
             SJMP CONTINUE       ;When we get back, we jump to CONTINUE label
CHECK3:      CJNE A,#50h,CONTINUE;If A is not 50h, we jump to CONTINUE label
             LCALL DEBOUNCE_KEY  ;If A is 50h, call the DEBOUNCE_KEY subroutine
CONTINUE:    {Code continues here};The rest of the program continues here
```

As you can see, the first line compares the Accumulator with 30h. If the Accumulator is not 30h it jumps to CHECK2 where the next comparison will be made. If the accumulator is 30h, however, program execution continues with the next instruction which calls the PROC_A subroutine. When the subroutine returns, the SJMP instruction causes the program to jump ahead to the CONTINUE label—thus bypassing the rest of the checks.

The code at label CHECK2 is the same as the first check. It first compares the Accumulator with 42h and then either branches to CHECK3 or calls the CHECK_LCD subroutine and jumps to CONTINUE. Finally, the code at CHECK3 does a final check for the value of 50h. This time there is no SJMP instruction following the call to DEBOUNCE_KEY since the next instruction is CONTINUE.

Code structures similar to the one shown above are very common in 8052 assembly language programs to execute certain code or subroutines based on the value of some register, in this case the Accumulator.

11.14 Less Than and Greater Than Comparison (CJNE)

Often it is necessary not to check whether a register is or isn't a certain value, but rather to determine whether a register is greater than or less than another register or value. As it turns out, the CJNE instruction—in combination with the carry flag—allows us to accomplish this.

When the CJNE instruction is executed not only does it compare *parameter1* to *parameter2* and branch if they aren't equal, it also sets or clears the carry bit based on which parameter is greater or less than the other.

1. If parameter1 < parameter2, the carry bit will be set (set to 1).
2. If parameter1 >= parameter2, the carry bit will be cleared (cleared to 0).

This is the way an assembly language program can do a greater than/less than comparison. For example, if the Accumulator holds some number and we want to know if it is less than or greater than 40h, the following code could be used:

```
            CJNE A,#40h,CHECK_LESS ;If A is not 40h, check if  < or > 40h
            LJMP A_IS_EQUAL     ;If A is 40h, jump to A_IS_EQUAL code
CHECK_LESS:  JC A_IS_LESS       ;If carry is set, A is less than 40h
A_IS_GREATER:       {Code}      ;Otherwise, it means A is greater than 40h
```

The code above first compares the Accumulator to 40h. If they are the same the program will fall through to the next line and jump to A_IS_EQUAL since we already know they are equal. If they aren't the same, execution will continue at CHECK_LESS. If the carry bit is set it means that the Accumulator was less than the second parameter (40h), so we jump to the label A_IS_LESS which will handle the "less than" condition. If the carry bit wasn't set then execution will fall through to A_IS_GREATER at which point you'd insert the code for the "greater than" condition.

NOTE: Keep in mind that CJNE will clear the carry bit if parameter1 is greater than *or equal* to parameter2. This means it is very important that you check to see if the values are equal before using the carry bit to determine less than/greater than. Otherwise, you might branch to the "greater than" condition when, in fact, the two parameters are equal.

11.15 Zero and Non-Zero Decisions (JZ/JNZ)

Sometimes it is useful to be able to simply determine if the Accumulator holds a zero or not. This could be done with a CJNE instruction, but since these types of tests are so common in software the 8052 instruction set provides two instructions for this purpose: **JZ** and **JNZ**.

JZ will jump to the given address or label if the Accumulator is zero. The instruction means "Jump if Zero."

JNZ will jump to the given address or label if the Accumulator is *not* zero. The instruction means "Jump if Not Zero".

For example:

```
      JZ ACCUM_ZERO       ;Jump to ACCUM_ZERO if the Accumulator = 0
      JNZ NOT_ZERO        ;Jump to NOT_ZERO if the Accumulator is not 0
```

Using JZ and/or JNZ is much easier and faster than using CJNE if all you want to test for is a zero/non-zero value in the Accumulator.

NOTE: Other non-8052 architectures have a "Zero Flag" that is set by instructions and the zero-test instruction tests that flag, not the Accumulator. The 8052, however, has no Zero Flag and JZ and JNZ both test the *value of the Accumulator*, *not* the status of any flag.

11.16 Performing Additions (ADD, ADDC)

The **ADD** and **ADDC** instructions provide a way to perform 8-bit addition. All addition involves adding some number or register to the Accumulator and leaving the result in the Accumulator. The original value in the Accumulator is always overwritten with the result of the addition.

```
ADD A,#25h   ;Add 25h to whatever value is in the Accumulator
ADD A,40h    ;Add contents of Internal RAM address 40h to Accumulator
ADD A,R4     ;Add the contents of R4 to the Accumulator
ADDC A,#22h  ;Add 22h to the Accumulator, plus carry bit
```

The ADD and ADDC instructions are identical except that ADD will only add the accumulator and the specified value or register whereas ADDC will also add the carry bit. The difference between the two, and the use of both, can be seen in the following code.

This code assumes that a 16-bit number is in Internal RAM address 30h (high byte) and address 31h (low byte). The code will add 1045h to the number leaving the result in addresses 32h (high byte) and 33h (low byte).

```
MOV A,31h    ;Move value from IRAM address 31h (low byte) to Accumulator
ADD A,#45h   ;Add 45h to the Accumulator (45h is low byte of 1045h)
MOV 33h,A    ;Move the result from Accumulator to IRAM address 33h
MOV A,30h    ;Move value from IRAM address 30h (hi byte) to Accumulator
ADDC A,#10h  ;Add 10h to the Accumulator (10h is the high byte of 1045h)
MOV 32h,A    ;Move result from Accumulator to Internal RAM address 32h
```

This code first loads the accumulator with the low byte of the original number from Internal RAM address 31h. It then adds 45h to it. Since the ADD instruction is used, it doesn't matter what the carry bit holds. The result is moved to 33h and the high byte of the original address is moved to the Accumulator from address 30h. The ADDC instruction then adds 10h to it, plus any carry that might have occurred in the first ADD step.

Both ADD and ADDC will set the carry flag if an addition of unsigned integers results in an overflow that can't be held in the accumulator, or will clear the carry flag if the . For example, if the Accumulator holds the value F0h and the value 20h is added to it, the Accumulator will hold the result of 10h and the carry bit will be set. The fact that the carry bit is set can be subsequently be used with the ADDC to add the carry into the next addition instruction.

The Auxiliary Carry (AC) bit is set if there is a carry from bit 3 and cleared otherwise. For example, if the Accumulator holds the value 2Eh and the value 05h is added to it, the Accumulator will then equal 33h as expected, but the AC bit will be set since the low nibble overflowed from Eh to 3h.

The Overflow (OV) bit is set if there is a carry out of bit 7 but not out of bit 6, or out of bit 6 but not out of bit 7. This is used in the addition of signed numbers to indicate that a negative number was produced as a result of the addition of two positive numbers, or that a positive number was produced by the addition of two negative numbers. For example, adding 20h to 70h (two positive numbers) would produce the value 90h. However, if the Accumulator is being treated as a signed number the value 90h would represent the number –10h. The fact that the OV bit was set means that the value in the Accumulator shouldn't really be interpreted as a negative number.

NOTE: Many other (non-8052) architectures only have a single type of ADD instruction—one that always includes the carry bit in the addition. The reason 8052 assembly language has two different types of ADD instructions is to avoid the need to start every addition calculation with a CLR C instruction. Using the ADD instruction is the same as using the CLR C instruction followed by the ADDC instruction.

11.17 Performing Subtractions (SUBB)

The **SUBB** instruction provides a way to perform 8-bit subtraction. All subtraction involves subtracting some number or register from the Accumulator and leaving the result in the Accumulator. The original value in the Accumulator is always overwritten with the result of the subtraction.

```
SUBB A,#25h   ;Subtract 25h from whatever value is in the Accumulator
SUBB A,40h    ;Subtract contents of IRAM address 40h from the Accumulator
SUBB A,R4     ;Subtract the contents of R4 from the Accumulator
```

The SUBB instruction always includes the carry bit in the subtract operation. This means if the Accumulator holds the value 38h and the carry bit is set, subtracting 6h will result in 31h (38h – 6h – carry bit).

NOTE: Since SUBB always includes the carry bit in its operation, it is necessary to always clear the carry bit (CLR C) before executing the first SUBB in a subtraction operation so that the prior status of the carry flag does not affect the instruction.

SUBB sets and clears the carry, auxiliary carry, and overflow bits in much the same way as the ADD and ADDC instructions.

SUBB will set the carry bit if the number being subtracted from the Accumulator is larger than the value in the Accumulator. In other words, the carry will be set if a borrow is needed for bit 7. Otherwise, the carry bit will be cleared.

The Auxiliary Carry (AC) bit will be set if a borrow is needed for bit 3, otherwise it is cleared.

The Overflow (OV) bit will be set if a borrow into bit 7 but not into bit 6, or into bit 6 but not into bit 7. This is used when subtracting signed integers. If subtracting a negative value from a positive value produces a negative number, OV will be set. Likewise, if subtracting a positive number from a negative number produces a positive number the OV flag will also be set.

11.18 Performing Multiplication (MUL)

In addition to addition and subtraction, the 8052 also offers the **MUL AB** instruction to multiply two 8-bit values. Unlike addition and subtraction, the MUL AB instruction always multiplies the contents of the Accumulator by the contents of the "B" register (SFR F0h). The result overwrites both the Accumulator and B, placing the low-byte of the result in the Accumulator and the high-byte of the result in B.

For example, to multiply 20h by 75h, the following code could be used:

```
MOV A,#20h    ;Load Accumulator with 20h
MOV B,#75h    ;Load B with 75h
MUL AB        ;Multiply A by B
```

The result of 20h x 75h is 0EA0h, thus after the above MUL instruction the Accumulator would hold the low-byte of the answer (A0h) and B would hold the high-byte of the answer (0Eh). The original values of the Accumulator and B are overwritten.

If the result is greater than 255 the Overflow (OV) bit will be set, otherwise it will be cleared. The carry bit is always cleared and the auxiliary carry (AC) bit is unaffected.

NOTE: You may multiply any two 8-bit values using MUL AB and obtain a result that will fit in the 16-bits available for the result in A and B. This is because the largest possible multiplication would be FFh X FFh which would result in FE01h which comfortably fits into the 16-bit space. It is not possible to overflow a 16-bit result space with two 8-bit multipliers.

11.19 Performing Division (DIV)

The last of the basic mathematics functions offered by the 8052 is the **DIV AB** instruction. This instruction, as the name implies, divides the Accumulator by the value held in the B register. Like the MUL instruction, this instruction *always* uses the Accumulator and B registers. The integer (whole-number) portion of the answer is placed in the accumulator and any remainder is placed in the B register. The original values of the Accumulator and B are overwritten.

For example, to divide F3h by 13h, the following code could be used:

```
MOV A,#0F3h   ;Load Accumulator with F3h
MOV B,#13h    ;Load B with 13h
DIV AB        ;Divide A by B
```

The result of F3h / 13h is 0Ch with remainder 0Fh, thus after this DIV instruction the Accumulator will hold the value 0Ch and B will hold the value 0Fh.

The Carry bit and the Overflow bits are both cleared by DIV, unless a division by zero is attempted in which case the Overflow bit is set. In the case of division by zero the result in the Accumulator and B after the instruction are undefined.

NOTE: While the MUL instruction takes two 8-bit values and multiplies them into a 16-bit value, the DIV instruction takes two 8-bit values and divides it into an 8-bit value and a remainder. The 8052 does not provide an instruction that will divide a 16-bit number.

CODE LIBRARY: You may find source code that includes 16-bit and 32-bit division in the Code Library at http://www.8052.com/codelib.phtml.

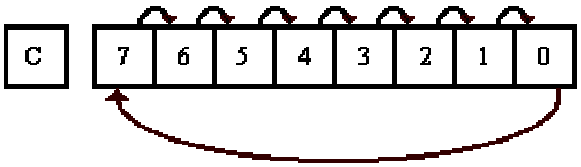
11.20 Shifting Bits (RR, RRC, RL, RLC)

The 8052 offers four instructions which are used to shift the bits in the accumulator to the left or right by one bit: **RR A**, **RRC A**, **RL A**, **RLC A**. There are two instructions that shift bits to the right, RR A and RRC A, and two that shift bits to the left, RL A and RLC A. The RRC and RLC instructions are different in that they rotate bits through the carry bit, whereas RR and RL don't involve the carry bit.

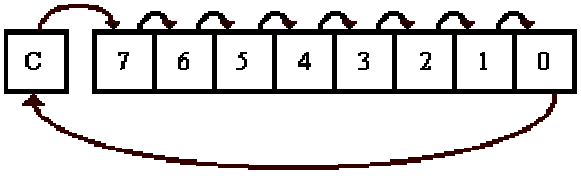
```
RR A   ;Rotate accumulator one bit to right, bit 0 is rotated into bit 7
RRC A  ;Rotate accumulator to right, bit 0 is rotated into carry,
       ;carry into bit 7
RL A   ;Rotate accumulator one bit to left, bit 7 is rotated into bit 0
RLC A  ;Rotate the accumulator to the left, bit 7 is rotated into
       ;carry, carry into bit 0
```

The following illustrations show how each of the instructions manipulates the eight bits of the Accumulator and the carry bit.

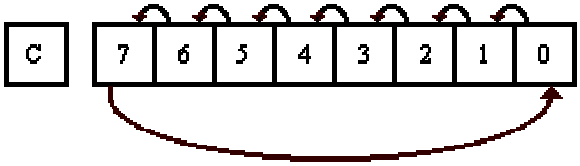
RR A - Rotate accumulator right one bit



RRC A - Rotate accumulator right one bit, with carry



RL A - Rotate accumulator left one bit



RLC A - Rotate accumulator left one bit, with carry

Using the shift instructions is, obviously, useful for bit manipulations. They can also be used, however, to quickly multiply or divided by multiples of two.

For example, if you would like to multiply the Accumulator by two, you could go about this two ways:

```
MOV B,#02h    ;Load B with 2
MUL AB        ;Multiply Accumulator by B (2), leaving low
              ;byte in Accumulator
```

Or you could simply use the RLC instruction:

```
CLR C         ;Make sure carry bit is initially clear
RLC A         ;Rotate left, multiplying by two
```

This may look like the same amount of work, but it isn't to the MCU. The first approach requires 4 bytes of program memory and takes 6 instruction cycles whereas the second approach requires only 2 bytes of program memory and 2 instruction cycles. So the RLC approach requires half as much memory and is 3 times as fast.

11.21 Bit-wise Logical Instructions (ANL, ORL, XRL)

The 8052 instruction set offers three instructions to perform the three most common types of bit-level logic which are "Logical And" (**ANL**), "Logical OR" (**ORL**), and "Logical Exclusive OR" (**XRL**). The instructions are capable of operating on the Accumulator or an Internal RAM address.

Some examples of these instructions are:

```
ANL A,#35h    ;Performs logical AND between Accumulator and 35h,
              ;result in Accumulator
ORL 20h,A     ;Performs logical OR between IRAM 20h and
              ;Accumulator, result in IRAM 20h
XRL 25h,#15h  ;Performs logical Exclusive OR between IRAM 25h and 15h
```

ANL (Logical AND) looks at each bit of *parameter1* and compares it to the same bit in *parameter2*. If the bit is set in both parameters, the bit remains set—otherwise the bit is cleared. The result is left in *parameter1*.

ORL (Logical OR) looks at each bit of *parameter1* and compares it to the same bit in *parameter2*. If the bit is set in either parameter, the bit remains set—otherwise the bit is cleared. The result is left in *parameter1*.

XRL (Logical Exclusive OR) looks at each bit of *parameter1* and compares it to the same bit in *parameter2*. If the bit is set in one of the two parameters, the bit is set—otherwise the bit is cleared. This means if the bit is set in both parameters it will be cleared. If it is set in one of the two parameters it will remain set. If it is clear in both parameters it will remain clear. The result is left in *parameter1*.

The following tables show the result of each of these logical instructions when applied to each possible bit combination.

ANL	0	1
0	0	0
1	0	1

ORL	0	1
0	0	1
1	1	1

XRL	0	1
0	0	1
1	1	0

Most of the logical bit-wise instructions affect entire 8-bit memory registers. However, the following instructions are available to perform logical operations on the carry bit. The result of these instructions is always left in the carry bit and the other bit is left unchanged.

ANL C,*bit*: This instruction will perform a logical AND between the Carry bit and the specified *bit*. If both bits are set, the carry bit will remain set. Otherwise the carry bit is cleared.

ANL C,*/bit*: This instruction performs a logical AND between the Carry bit and the *complement* of the specified *bit*. That means if the *bit* is set, the carry bit will be ANDed as if it were clear. If the specified *bit* is clear, it will be ANDed with the carry bit as if it were set.

ORL C,*bit*: This instruction will perform a logical OR between the Carry bit and the specified *bit*. If either the carry bit or *bit* is set, the carry bit will be set. If neither bit is set, the carry bit will be cleared.

ORL C,*/bit*: This instruction performs a logical OR between the Carry bit and the *complement* of the specified *bit*. That means if the *bit* is set, the carry bit will be ORed as if it were clear. If the specified *bit* is clear, it will be ORed with the carry bit as if it were set.

NOTE: There is no XRL that operates on the Carry bit and another bit. Only the ANL and ORL logical instructions are supported with the Carry bit.

11.22 Exchanging Register Values (XCH)

Very often you will want to swap the value of the Accumulator with the value of another SFR or Internal RAM address. The **XCH** instruction allows you to do this quickly and without using additional temporary holding variables.

XCH will take the value of the Accumulator and write it to the specified SFR or Internal RAM address while at the same time writing the original value of that SFR or Internal RAM address to the Accumulator.

For example:

```
MOV A,#25h   ;Accumulator now holds 25h
MOV 60h,#45h ;Internal RAM 60h now holds 45h
XCH A,60h    ;Accumulator now holds 45, IRAM 60h now holds 25h
```

11.23 Swapping Accumulator Nibbles (SWAP)

In some cases it can be useful to swap the nibbles of the Accumulator. A nibble is 4 bits, so there are two nibbles in the Accumulator. The "high" nibble consists of bits 4 through 7 while the "low" nibble consists of bits 0 through 3.

The **SWAP A** instruction will swap the two nibbles of the Accumulator. For example, if the Accumulator holds the value 56h, the SWAP instruction will convert it to 65h. Likewise, F7h will be converted into 7Fh.

NOTE: The SWAP A instruction is identical to executing four RL A instructions or four RR A instructions.

11.24 Exchanging Nibbles Between Accumulator and Internal RAM (XCHD)

The **XCHD** instruction swaps the low nibble of the Accumulator with the low nibble of the register or Internal RAM address specified in the instruction.

For example, if R0 holds 87h and the Accumulator holds 24h, then the **XCHD R0** instruction will result in the Accumulator holding 27h and R0 holding 84h. The low nibbles of the two were simply exchanged.

I personally have never used this instruction, but presumably it is useful in some situations since 11 opcodes of the 8052 instruction set are devoted to it.

11.25 Adjusting Accumulator for BCD Addition (DA)

DA A is a very useful instruction if you are doing BCD-encoded addition.

First of all, BCD stands for Binary Coded Decimal. BCD is a form of expressing two decimal digits in a single 8-bit byte. If you express any 8-bit value in hexadecimal, it can be expressed as a number between 00 and FF. Obviously, it is possible to express all normal decimal numbers between 0 and 99 in hexadecimal format so that, printed as hexadecimal, they appear to be decimal numbers.

For example, the decimal digits "00" would be represented in BCD as, not surprisingly, 00h. The decimal digits "09" would be represented in BCD as 09h. The decimal digits "10", however, would be represented in BCD as 10h—but note that 10h is actually 16 (decimal). That's because in BCD we don't use the hex values A, B, C, D, E, and F. Thus we jump from 09h to 10h.

This is all fine and good, but what happens when we want to add two BCD numbers together? For example, what happens if we add 38 to 25? Obviously, in normal decimal math, 38 + 25 = 63. Ideally, doing the same addition on BCD encoded values would have the same result.

But 38 encoded as BCD is 38h and 25 encoded as BCD is 25h. 38h + 25h = 5Dh. Obviously the result no longer looks like a decimal value—and that's not surprising since BCD doesn't use the values A, B, C, D, E, and F.

What DA A does is automatically "adjusts" the Accumulator after the addition of two BCD values. In the above example, executing DA A when the Accumulator holds 5Dh will result in the Accumulator being adjusted to 63h, thus "righting" our rather strange addition.

The details of how DA A works and why are not extremely important to this tutorial and would tend to confuse things rather than explain them. If you plan on doing BCD addition I invite you to investigate this instruction further. For the majority that will not be doing BCD addition, you can safely ignore this instruction.

11.26 Using the Stack (PUSH/POP)

The stack, as with any processor, is an area of memory that can be used to store information temporarily, including the return address for returning from subroutines that are called by ACALL or LCALL. The 8052 automatically handles the stack when you make an ACALL or LCALL, as well as when you return with the RET instruction. The stack is also handled automatically when an interrupt service routine is triggered by an interrupt, and when you return from the ISR with the RETI instruction.

You can also use the stack for your own purposes and temporary storage by using the **PUSH** and **POP** instructions. The PUSH instruction will "push" a value onto the stack, and the POP instruction will "pop" the last value you pushed on the stack. You may save a value temporarily by PUSHing it onto the stack, and you may restore that value by POPping it.

NOTE: The stack operates on a Last In-First Out (LIFO) basis. This means if you PUSH the values 4, 5, and 6 (in that order), POPping them one at a time will return 6, 5, and then 4. The value most recently added to the stack is the first value that will come off when you execute a POP instruction.

An example using the PUSH and POP instruction is:

```
MOV A,#35h    ;Load the Accumulator with the value 35h
PUSH ACC      ;Push accumulator onto stack, Accumulator still holds 35h
ADD A,#40h    ;Add 40h to the Accumulator, Accumulator now holds 75h
POP ACC       ;Pop the accumulator from stack, Accumulator holds
              ;35h again
```

The above code is functionally useless—it doesn't do anything useful. But it does illustrate how to use PUSH and POP.

The code starts by assigning 35h to the Accumulator. It then PUSHes it onto the stack. We then add 40h to the Accumulator, just to change the Accumulator to something else. At this point the Accumulator holds 75h. Finally, we POP from the stack into the Accumulator. Since the last value pushed onto the stack was 35h, the POP restores the value of the Accumulator to 35h.

NOTE: When PUSHing or POPping the Accumulator, you must refer to it as ACC since that is the memory location of the SFR. You may *not* assemble the instruction PUSH A nor POP A—both of these will result in an assemble-time error in most, if not all, 8052 assemblers.

When using PUSH, the SFR or Internal RAM address that follows the PUSH instruction is the value that will be PUSHed onto the stack. For example, **PUSH ACC** will push the value of the Accumulator onto the stack. **PUSH 60h** will push the value of Internal RAM address 60h onto the stack.

Likewise, the Internal RAM address or SFR that follows a POP instruction indicates where the value should be stored when it is POPped from the stack. For example, **POP ACC** will pop the next value off the stack and into the Accumulator. **POP 40h** will pop the next value off the stack and into Internal RAM address 40h.

The stack itself resides in Internal RAM and is "managed" by the SP (Stack Pointer) SFR. SP will always point to the Internal RAM address from which the next POP instruction should obtain the data.

1. **POP** will return the value of the Internal RAM address pointed to by SP, then decrement SP by 1.
2. **PUSH** will increment SP by 1, then store the value at the IRAM address then pointed to by SP.

SP is initialized to 07h when an 8052 is first powered-up. That means the stack will begin at 08h and start growing from there. If you push 16 values onto the stack, for example, the stack will occupy addresses 08h through 17h.

Using the stack can be both useful and powerful, but it can also be dangerous when incorrectly used. Remember that the stack is also used by the 8052 to remember the return addresses of subroutines and interrupts. If you modify the stack incorrectly it is very easy to cause your program to crash or to behave in very unexpected ways.

When using the stack, all but advanced stack users should observe the following recommendations:

1. When using the stack from within a subroutine or interrupt service routine, be sure you have one POP instruction for every PUSH instruction. If the number of POPs and PUSHes aren't the same, your program will probably end up crashing.
2. When using PUSH, be sure you always POP that value off the stack—even if you're not in a subroutine.
3. Be sure you don't jump over the section of code that POPs a value off the stack. A common error is to PUSH a value onto the stack and then execute a conditional instruction that jumps over the instruction that POPs that value off. This results in an unbalanced stack and will probably end up crashing your program. Remember, not only must you have a POP instruction for every PUSH, you must *execute* a POP instruction for every PUSH that is executed. Make sure your program doesn't jump over your POP instructions.
4. Always make sure you use the RET instruction to return from subroutines and RETI instruction to return from interrupt service routines.
5. As a practice, only modify SP at the very beginning of your program in order to initialize it. Once you start using the stack or making calls to subroutines you should *not* modify SP.
6. Make sure your stack has enough room. For example, the stack will start by default at address 08h. If you have a variable at Internal RAM address 20h then your stack has only 24 bytes available to it, from 08h through 1Fh. If your stack is 24 bytes long and another value is pushed onto the stack or another subroutine is called, your variable at 20h will be overwritten.
7. Remember that the stack is a LIFO stack, which means you must POP values off the stack in the reverse order that you PUSHed them onto the stack.

Keep in mind, too, that the 8052 can only use Internal RAM for its stack. Even if you have 64k of External RAM, the 8052 can only use its 256 bytes of Internal RAM for the stack. This means you should use the stack very sparingly.

11.27 Setting the Data Pointer DPTR (MOV DPTR)

The next few instructions use the Data Pointer (DPTR), the 8052's only 16-bit register. DPTR is used to point to a RAM or ROM address when used with the instructions that will be explained below.

As described earlier, DPTR really is made up of two SFRs: DPH and DPL which hold the high and low bytes, respectively, of the 16-bit Data Pointer. But when DPTR is used to access memory the 8052 will treat DPTR as a single address.

To set the DPTR to a specific, the **MOV DPTR** instruction is used. This instruction lets you set both DPH and DPL in a single instruction. However, you can still modify DPTR by accessing DPH and DPL directly, as illustrated in the following examples:

```
MOV DPTR,#1234h     ;Sets DPTR to 1234h
MOV DPTR,#0F123h    ;Sets DPTR to F123h
MOV DPH,#40h        ;Sets DPTR high-byte to 40h (DPTR now 4023h)
MOV DPL,#56h        ;Sets DPTR low-byte to 56h (DPTR now 4056h)
```

As you can, the first two instructions set DPTR first to 1234h and then to F123h. The next example sets DPH to 40h, leaving DPTR's low-byte unchanged. Since the low byte is still 23h from the previous example, changing DPH to 40h will result in DPTR being equal to 4023h. Finally, we change the low-byte to 56h, leaving the high-byte unchanged. Since the high-byte was set to 40h in the previous example, setting the low-byte to 56h will leave the DPTR with a value of 4056h.

In other words, **MOV DPTR,#1567h** is the same as **MOV DPH,#15h** and **MOV DPL,#67h**. The advantage to using MOV DPTR is that it uses only 3 bytes of memory and 2 instruction cycles wheras the other method requires 6 bytes of memory and 4 instruction cycles.

11.28 Reading External RAM/Data Memory (MOVX)

The 8052 generally has 128 or 256 bytes of Internal RAM which is accessed with the MOV instruction, as described earlier. However, many projects will require more than 256 bytes of RAM. The 8052 has the ability of addressing up to 64k of "External RAM" which takes the form of additional, off-chip ICs.

The **MOVX** instruction is used to read and write to External RAM. The MOVX instruction has four forms:

1. **MOVX A,@DPTR**: Reads External RAM address DPTR into the Accumulator.
2. **MOVX A,@Ri**: Reads External RAM address pointed to by R0 or R1 into the Accumulator.
3. **MOVX @DPTR,A**: Sets External RAM address DPTR to the value of the Accumulator
4. **MOVX @Ri,A**: Sets the External RAM address held in R0 or R1 to the value of the Accumulator

The first two forms move data from External RAM into the Accumulator wheras the last two forms move data from the Accumulator into External RAM.

MOVX with DPTR: When using the forms of MOVX that use DPTR, DPTR will be used as a 16-bit memory address. The 8052 will automatically communicate with the off-chip RAM and obtain the value of that memory address and store it in the Accumulator (MOVX A,@DPTR), or will write the Accumulator to the off-chip RAM (MOVX @DPTR,A).

For example, to add 5 to the value contained in External RAM address 2356h the following code could be used:

```
MOV DPTR,#2356h     ;Set DPTR to 2356h
MOVX A,@DPTR        ;Read External RAM address 2356h into Accumulator
ADD A,#05h          ;Add 5 to the Accumulator
MOVX @DPTR,A        ;Write new value of Accumulator back to
                    ;External RAM 2356h
```

MOVX with @R0 or @R1: When using the forms of MOVX that use @R0 or @R1, R0 or R1 will be used to determine the address of External RAM to access. Since both R0 and R1 are 8-bit registers you can only use these forms of MOVX to access External RAM addresses 0000h through 00FFh, unless you take actions to control the high-byte of the address.

11.29 Reading Code Memory/Tables (MOVC)

It is often useful to be able to read code memory itself from within a program. This allows for the placement of data or tables in code memory to be read at run-time by the program itself. This is accomplished by the **MOVC** instruction.

The MOVC instruction comes in two forms: **MOVC A,@A+DPTR** and **MOVC A,@A+PC**. Both instructions move a byte of code memory into the Accumulator. The code memory address from which the byte is read depends on which of the two forms is used.

MOVC A,@A+DPTR will read the byte from the code memory address calculated by adding the current value of the Accumulator to that of DPTR. For example, if DPTR holds the value 1234h and the Accumulator holds the value 10h then the instruction would copy the value of code memory address 1244h into the Accumulator. This can be thought of as an "absolute" read since the byte will always be read from the address contained in the two registers Accumulator and DPTR. DPTR is initialized to point to the first byte of the table and the Accumulator is used as an offset into the table.

For example, perhaps we have a table of values that resides at 2000h in code memory. We need to write a subroutine that obtains one of those six values based on the value of the Accumulator. This could be coded as:

```
        MOV A,#04h          ;Set Accumulator to offset into the
                            ;table we want to read
        LCALL SUB           ;Call subroutine to read 4th byte of the table
        …
SUB:    MOV DPTR,#2000h     ;Set DPTR to the beginning of the value table
        MOVC A,@A+DPTR      ;Read the 5th byte from the table
        RET                 ;Return from the subroutine
```

MOVC A,@A+PC will read the byte from the code memory address calculated by adding the current value of the Accumulator to that of the Program Counter; that is, the address of the currently executing instruction. This can be thought of as a "relative" read since the address of code memory from which the byte will be read depends on where the MOVC instruction is found in memory. This form of MOVC is used when the data to be read immediately follows the code that is to read it.

For example, if instead of the data in the above example being located at code memory 2000h it were located right after the routine that read it, we could change the subroutine to:

```
SUB:    INC A           ;Increment Acc to account for RET instruction
        MOVC A,@A+PC    ;Get the data from the table
        RET             ;Return from subroutine
        DB 01h,02h,03h,04h,05h     ;The actual data table
```

Note that in the above example we first increment the Accumulator by 1. This is because the value of PC will be that of the instruction immediately following the MOVC instruction—in this case, the RET instruction. We don't want to get the RET opcode, rather we want the data that follows RET. Since the RET instruction requires one byte of code memory we need to INCrement the Accumulator 1 byte to "skip over" the RET instruction.

NOTE: The value that Accumulator must be incremented by is the number of bytes between the MOVC instruction and the first data of the table being read. For example, if the RET instruction above were replaced with an LJMP instruction, which is 3 bytes long, you would replace the INC A instruction with ADD A,#03h to increment the Accumulator by 3.

11.30 Using Jump Tables (JMP @A+DPTR)

A frequent method for quickly branching to many different areas in a program is the use of jump tables. For example, if we had to branch to different subroutines based on the value of the Accumulator we could accomplish this with the CJNE instruction that we have already covered:

```
                CJNE A,#00h,CHECK1  ;If it's not zero, jump to CHECK1
                AJMP SUB0           ;Go to SUB0 subroutine
CHECK1:         CJNE A,#01h,CHECK2  ;If it's not 1, jump to CHECK2
                AJMP SUB1           ;Go to SUB1 subroutine
CHECK2:
```

The above code will work, but if each additional possible value will increase the size of the program by 5 bytes—3 bytes for the CJNE instruction and 2 bytes for the AJMP instruction.

A more efficient way is to create a "jump table" by using the **JMP @A+DPTR** instruction. Like the MOVC @A+DPTR, this instruction will calculate an address by summing the Accumulator and DPTR and then jump to that address. So if DPTR holds 2000h and the Accumulator holds 14h, the JMP instruction will jump to 2014h.

Consider the following code:

```
            RL A                    ;Rotate Accumulator left, multiply by 2
            MOV DPTR,#JUMP_TABLE    ;Load DPTR with address of jump table
            JMP  @A+DPTR            ;Jump to the corresponding address
JUMP_TABLE: AJMP SUB0               ;Jump table entry to SUB0
            AJMP SUB1
```

This code first takes the value of the Accumulator and multiplies it by 2 by shifting the Accumulator to the left by one bit. Since each AJMP entry in JUMP_TABLE is 2-bytes long we must first multiply the Accumulator by two.

The code then loads the DPTR with the address of the JUMP_TABLE and proceeds to JMP to the address of the Accumulator plus DPTR. Since we already know that we want to jump to the offset indicated by the Accumulator, no additional checks are necessary. We jump directly into the table that jumps to our subroutine. Each additional entry in the jump table will require only 2 additional bytes (2 bytes for each AJMP instruction).

NOTE: It's almost always a good idea to use a jump table if you have 2 or more choices based on a zero-based index. A jump table with just two entries, like the above example, will save 1 byte of memory over using the CJNE approach, and you will save 3 bytes of memory for each additional entry.

NOTE: An additional benefit of jump tables is that the execution time is *always the same*. Using a sequence of CJNE instructions means that the code will execute faster if the value being searched for is at the beginning of the CJNE sequence and will execute slower if it is at the end of the CJNE sequence. A jump table always takes the same amount of time to execute and, thus, can be very helpful in time-critical code which must take a specific amount of time to execute.

Chapter 12: 16-Bit Mathematics with the 8051

The 8052 is an 8-bit microcontroller. This basically means that each machine language opcode in its instruction set consists of a single 8-bit value. This permits a maximum of 256 instruction codes (of which 255 are actually used in the 8052 instruction set).

The 8052 also works almost exclusively with 8-bit values. The Accumulator is an 8-bit value, as is each register in the Register Banks, the Stack Pointer (SP), and each of the many Special Function Registers (SFRs) that exist in the architecture. In reality, the only values that the 8052 handles that are truly 16-bit values are the Program Counter (PC) that internally indicates the next instruction to be executed, and the Data Pointer (DPTR) which the user program may utilize to access external RAM as well as directly access code memory. Other than these two registers, the 8052 works exclusively with 8-bit values.

For example, the ADD instruction will add two 8-bit values to produce a third 8-bit value. The SUBB instruction subtracts an 8-bit value from another 8-bit value and produces a third 8-bit value. The MUL instruction will multiply two 8-bit values and produce a 16-bit value.

Programming Tip: It could be said that the MUL instruction is a 16-bit math instruction since it produces a 16-bit answer. However, its inputs are only 8-bit. The result is 16-bits out of necessity since any multiplication with two operands greater than the number 16 will produce a 16-bit result. Thus, for the MUL operation to have any value at all it was absolutely necessary to produce a 16-bit result.

As we can see, the 8052 provides us with a number of instructions aimed at performing mathematical calculations. Unfortunately, they are all work with 8-bit input values--and we often find ourselves working with values that simply cannot be expressed in 8-bits.

This tutorial will discuss techniques that allow the 8052 developer to work with 16-bit values in the 8052's 8-bit architecture. While we will only discuss 16-bit mathematics, the techniques can be extended to any number of bits (24-bit, 32-bit, 64-bit, etc.). It's just a matter of expanding the code to support the additional bytes. The algorithms remain the same.

These tutorials will explain how to perform 16-bit addition, subtraction, and multiplication with the 8052. For the time being, 16-bit division is outside the scope of this tutorial.

Programming Tip: Compared to addition, subtraction, and multiplication, division is a relatively complicated process. For the time being 16-bit division will not be discussed because the author has not had a need to develop such routines, nor an opportunity to analyze the process in performing the calculation. If you have developed a routine that allows a 16-bit value to be divided by another 16-bit value and would like to contribute the code to 8052.com, along with a tutorial similar to those found in these sections, please contact us at **8052com@8052.com**.

12.1 How did we learn math in primary school?

Before jumping into multi-byte mathematics in machine language, let's quickly review the mathematics we learned as children. For example, we learned to add two numbers, say 156 + 248, as follows:

	100's	10's	1's
	1	5	6
+	2	4	8
=	4	0	4

How do we calculate the above? We start in the 1's column, adding 6 + 8 = 14. Since 14 can't fit in a single column, we leave 4 in the 1's column and carry the 1 to the 10's column. We then add 5 + 4 = 9, add the 1 we carried, to get 10. Again, 10 does not fit in a single column. So we leave the 0 in the 10's column and carry the 1 to the 100's column. Finally, we add 1 + 2 = 3, add the 1 we carried to get 4, which is our final answer in the 100's column. The final answer, thus, is 404.

It is important to remember this, when working with multi-byte math, because the process is going to be the same. Let's start by doing 16-bit addition.

12.2 16-bit Addition

16-bit addition is the addition of two 16-values. First, we must recognize that the addition of two 16-bit values will result in a value that is, at most, 17 bits long. Why is this so? The largest value that can fit in 16-bits is 256 * 256 - 1 = 65,535. If we add 65,535 + 65,535, we get the result of 131,070. This value fits in 17 bits. Thus when adding two 16-bit values, we will get a 17-bit value. Since the 8051 works with 8-bit values, we will use the following statement: Adding two 16-bit values results in a 24-bit value. Of course, 7 of the highest 8 bits will never be used--but we will have our entire answer in 3 bytes. Also keep in mind that we will be working with *unsigned integers*.

Programming Tip: Another option, instead of using 3 full bytes for the answer, is to use 2 bytes (16-bits) for the answer, and the carry bit (C), to hold the 17th bit. This is perfectly acceptable, and probably even preferred. The more advanced programmer will understand and recognize this option, and be able to make use of it. However, since this is an introduction to 16-bit mathematics it is our goal that the answer produced by the routines be in a form that is easy for the reader to utilize, once calculated. It is our belief that this is best achieved by leaving the answer fully expressed in 3 8-bit values.

Let's consider adding the following two decimal values: 6724 + 8923. The answer is, of course, 15647. How do we go about adding these values with the 8051? The first step is to always work with hexadecimal values: convert the two values you wish to add to hexadecimal. In this case, that is equivalent to the following hexadecimal addition: 1A44 + 22DB.

How do we add these two numbers? Let's use the exact same method we used in primary school, and in the section above:

	256's	1's
	1A	44
+	22	DB
=	3D	1F

First, notice the difference. We are no longer working with a 1's, 10's, and 100's columns. We are just working with two columns: The 1's column and the 256's column. In familiar computer terms: We're working with the low byte (the 1's column) and the high byte (the 256's column). However, the process is exactly the same.

First we add the values in the 1's column (low byte): 44 + DB = 11F. Only a 2-digit hexadecimal value can fit in a single column, so we leave the 1F in the low-byte column, and carry the 1 to the high-byte column. We now add the high bytes: 1A + 22 = 3C, plus the 1 we carried from the low-byte column. We arrive at the value 3D.

Thus, our completed answer is 3D1F. If we convert 3D1F back to decimal, we arrive at the answer 15647. This matches with the original addition we did in decimal. The process works. Thus the only challenge is to code the above process in 8052 assembly language. As it turns out, this is incredibly easy.

We'll use the following table to explain how we're going to do the addition:

	65536's	256's	1's
		R6	R7
+		R4	R5
=	R1	R2	R3

Since we're adding 16-bit values, each value requires two 8-bit registers. Essentially, the first value to add will be held in R6 and R7 (the high byte in R6 and the low byte in R7) while the second value to add will be held in R4 and R5 (the high byte in R4 and the low byte in R5). We will leave our answer in R1, R2, and R3.

Programming Tip: Remember that we mentioned above that the sum of two 16-bit values is a 17-bit value. In this case, we'll be using 24-bits (R1, R2, and R3) for our answer, even though we'll never use more than 1 bit of R1.

Let's review the steps involved in adding the values above:

1. Add the low bytes R7 and R5, leave the answer in R3.
2. Add the high bytes R6 and R4, adding any carry from step 1, and leave the answer in R2.
3. Put any carry from step 2 in the final byte, R1.

We'll now convert the above process to assembly language, step by step.

Step 1: Add the low bytes R7 and R5, leave the answer in R3.

```
MOV A,R7      ;Move the low-byte into the accumulator
ADD A,R5      ;Add the second low-byte to the accumulator
MOV R3,A      ;Move the answer to the low-byte of the result
```

Step 2: Add the high bytes R6 and R4, adding any carry from step 1, and leave the answer in R2.

```
MOV A,R6      ;Move the high-byte into the accumulator
ADDC A,R4     ;Add second high-byte to the accumulator, plus carry.
MOV R2,A      ;Move the answer to the high-byte of the result
```

Step 3: Put any carry from step 2 in the final byte, R1.

```
MOV A,#00h    ;By default, the highest byte will be zero.
ADDC A,#00h   ;Add zero, but this will add one if there was a
              ;carry from step 2.
MOV MOV R1,A ;Move the answer to the highest byte of  the result
```

That's it! Combining the code from the three steps, we come up with the following subroutine:

```
ADD16_16:
       ;Step 1 of the process
       MOV A,R7      ;Move the low-byte into the accumulator
       ADD A,R5      ;Add the second low-byte to the accumulator
       MOV R3,A      ;Move the answer to the low-byte of the result
       ;Step 2 of the process
       MOV A,R6      ;Move the high-byte into the accumulator
       ADDC A,R4     ;Add the second high-byte to the accumulator, plus carry.
       MOV R2,A      ;Move the answer to the high-byte of the result
       ;Step 3 of the process
       MOV A,#00h    ;By default, the highest byte will be zero.
       ADDC A,#00h   ;Add zero, but this will add one if there was
                     ;a carry from step 2.
       MOV MOV R1,A ;Move the answer to the highest byte of  the result
       RET           ;Return - answer now resides in R1, R2, and R3.
```

And to call our routine to add the two values we used in the example above, we'd use the code:

```
       ;Load the first value into R6 and R7
       MOV R6,#1Ah
       MOV R7,#44h

       ;Load the first value into R6 and R7
       MOV R4,#22h
       MOV R5,#0DBh

       ;Call the 16-bit addition routine
       LCALL ADD16_16
```

12.3 16-bit Subtraction

16-bit subtraction is the subtraction of one 16-bit value from another. A subtraction of this nature results in another 16-bit value. Why? The number 65535 is a 16-bit value. If we subtract 1 from it, we have 65534 which is also a 16-bit value. Thus any 16-bit subtraction will result in another 16-bit value.

Let's consider the subtraction of the following two decimal values: 8923 - 6905. The answer is 2018. How do we go about subtracting these values with the 8051? As is the case with addition, the first step is to convert the expression to hexadecimal. The above decimal subtraction is equivalent to the following hexadecimal subtraction: 22DB - 1AF9.

Again, we'll go back to the way we learned it in primary school:

	256's	1's
	22	DB
-	1A	F9
=	7	E2

First we subtract the second value in the 1's column (low byte): DB - F9. Since F9 is greater than DB, we need to "borrow" from the 256's column. Thus we actually perform the subtraction 1DB - F9 = E2. The value E2 is what we leave in the 1's column.

Now we must perform the subtraction 22 - 1A. However, we must remember that we "borrowed" 1 from the 256's column, so we must subtract an additional 1. So the subtraction for the 256's column becomes 22 - 1A - 1 = 7, which is the value we leave in the 256's column.

Thus our final answer is 07E2. If we conver this back to decimal, we get the value 2018, which coincides with the math we originally did in decimal.

As we did with addition, we'll use a small table to help us conver the above process to 8051 assembly language:

	256's	1's
	R6	R7
-	R4	R5
=	R2	R3

Since we're subtracting 16-bit values, each value requires two 8-bit registers. Essentially, the value to be subtracted from will be held in R6 and R7 (the high byte in R6 and the low byte in R7) while the value to be subtracted will be held in R4 and R5 (the high byte in R4 and the low byte in R5). We will leave our answer in R2, and R3.

Let's review the steps involved in subtracting the values above:

1. Subtract the low bytes R5 from R7, leave the answer in R3.
2. Subtract the high byte R4 from R6, less any borrow, and leave the answer in R2.

We'll now convert the above process to assembly language, step by step.

Step 1: Subtract the low bytes R5 from R7, leave the answer in R3.

```
MOV A,R7            ;Move the low-byte into the accumulator
CLR C               ;Always clear carry before first subtraction
SUBB A,R5           ;Subtract the second low-byte from the accumulator
MOV R3,A            ;Move the answer to the low-byte of the result
```

Step 2: Subtract the high byte R4 from R6, less any borrow, and leave the answer in R2.

```
MOV A,R6            ;Move the high-byte into the accumulator
SUBB A,R4           ;Subtract the second high-byte from the accumulator
MOV R2,A            ;Move the answer to the low-byte of the result
```

Programming Tip: The SUBB instruction always subtracts the second value in the instruction from the first, less any carry. While there are two versions of the ADD instruction (ADD and ADDC), one of which ignores the carry bit, there is no such distinction with the SUBB instruction. This means before you perform the first subtraction, you must always be sure to clear the carry bit. Otherwise, if the carry bit happens to be set you'll end up subtracting it from your first column value -- which would be incorrect.

Combining the code from the two steps above, we come up with the following subroutine:

```
SUBB16_16:
      ;Step 1 of the process
      MOV A,R7            ;Move the low-byte into the accumulator
      CLR C               ;Always clear carry before first subtraction
      SUBB A,R5           ;Subtract the second low-byte from the accumulator
      MOV R3,A            ;Move the answer to the low-byte of the result
      ;Step 2 of the process
      MOV A,R6            ;Move the high-byte into the accumulator
      SUBB A,R4           ;Subtract the second high-byte from the accumulator
      MOV R2,A            ;Move the answer to the low-byte of the result
```

```
;Return - answer now resides in R2, and R3.
RET
```

And to call our routine to add the two values we used in the example above, we'd use the code:

```
;Load the first value into R6 and R7
MOV R6,#22h
MOV R7,#0DBh

;Load the first value into R6 and R7
MOV R4,#1Ah
MOV R5,#0F9h

;Call the 16-bit subtraction routine
LCALL SUBB16_16
```

12.4 16-bit Multiplication

16-bit multiplication is the multiplication of two 16-bit value from another. Such a multiplication results in a 32-bit value.

Programming Tip: Any multiplication results in an answer that has a size equal to the sum of te number of bits in each of the multiplicands. For example, multiplying an 8-bit value by a 16-bit value results in a 24-bit value (8 + 16). A 16-bit value multiplied by another 16-bit value results in a 32-bit value (16 + 16), etc.

For the sake of example, let's multiply 25,136 by 17,198. The answer is 432,288,928. As with both addition and subtraction, let's first convert the expression into hexadecimal: 6230h x 432Eh.

Once again, let's arrange the numbers in columns as we did in primary school to multiply numbers, although now the grid becomes more complicated. The green section represents the original two values. The yellow section represents the intermediate calculations obtained by multiplying each byte of the original values. The red section of the grid indicates our final answer, obtained by summing the columns in the yellow area.

	Byte 4	Byte 3	Byte 2	Byte 1
			62	30
X			43	2E
	====	====	====	====
			08	A0
		11	9C	
		0C	0A	
	19	A6		
	====	====	====	====
	19	C4	34	A0

Remember how we did this in elementary school? First we multiply 2Eh by 30h (byte 1 of both numbers), and place the result directly below. Then we multiply 2Eh by 62h (byte 1 of the bottom number by byte 2 of the upper number). This result is lined up such that the right-most column ends up in byte 2. Next we multiply 43h by 30h (byte 2 of the bottom number by byte 1 of the top number), again lining up the result so that the right-most column ends up in byte 2. Finally, we multiply 43h by 62h (byte 2 of both numbers)

and position the answer such that the right-most column ends up in byte 3. Once we've done the above, we add each column, with appropriate carries, to arrive at the final answer.

Our process in assembly language will be identical. Let's use our now-familiar grid to help us get an idea of what we're doing:

	Byte 4	Byte 3	Byte 2	Byte 1
			R6	R7
*			R4	R5
=	R0	R1	R2	R3

Thus our first number will be contained in R6 and R7 while our second number will be held in R4 and R5. The result of our multiplication will end up in R0, R1, R2 and R3. At 8-bits per register, these four registers give us the 32 bits we need to handle the largest possible multiplication. Our process will be the following:

1. Multiply R5 by R7, leaving the 16-bit result in R2 and R3.
2. Multiply R5 by R6, adding the 16-bit result to R1 and R2.
3. Multiply R4 by R7, adding the 16-bit result to R1 and R2.
4. Multiply R4 by R6, adding the 16-bit result to R0 and R1.

We'll now convert the above process to assembly language, step by step.

Step 1. Multiply R5 by R7, leaving the 16-bit result in R2 and R3.

```
MOV A,R5      ;Move the R5 into the Accumulator
MOV B,R7      ;Move R7 into B
MUL AB        ;Multiply the two values
MOV R2,B      ;Move B (the high-byte) into R2
MOV R3,A      ;Move A (the low-byte) into R3
```

Step 2. Multiply R5 by R6, adding the 16-bit result to R1 and R2.

```
MOV A,R5      ;Move R5 back into the Accumulator
MOV B,R6      ;Move R6 into B
MUL AB        ;Multiply the two values
ADD A,R2      ;Add the low-byte into the value already in R2
MOV R2,A      ;Move the resulting value back into R2
MOV A,B       ;Move the high-byte into the accumulator
ADDC A,#00h   ;Add zero (plus the carry, if any)
MOV R1,A      ;Move the resulting answer into R1
MOV A,#00h    ;Load the accumulator with  zero
ADDC A,#00h   ;Add zero (plus the carry, if any)
MOV R0,A      ;Move the resulting answer to R0.
```

Step 3. Multiply R4 by R7, adding the 16-bit result to R1 and R2.

```
MOV A,R4      ;Move R4 into the Accumulator
MOV B,R7      ;Move R7 into B
MUL AB        ;Multiply the two values
ADD A,R2      ;Add the low-byte into the value already in R2
MOV R2,A      ;Move the resulting value back into R2
MOV A,B       ;Move the high-byte into the accumulator
ADDC A,R1     ;Add the current value of R1 (plus any carry)
MOV R1,A      ;Move the resulting answer into R1.
MOV A,#00h    ;Load the accumulator with zero
ADDC A,R0     ;Add the current value of R0 (plus any carry)
MOV R0,A      ;Move the resulting answer to R1.
```

Step 4. Multiply R4 by R6, adding the 16-bit result to R0 and R1.

```
       MOV A,R4      ;Move R4 back into the Accumulator
       MOV B,R6      ;Move R6 into B
       MUL AB        ;Multiply the two values
       ADD A,R1      ;Add the low-byte into the value already in R1
       MOV R1,A      ;Move the resulting value back into R1
       MOV A,B       ;Move the high-byte into the accumulator
       ADDC A,R0     ;Add it to the value already in R0 (plus any carry)
       MOV R0,A      ;Move the resulting answer back to R0
```

Combining the code from the two steps above, we come up with the following subroutine:

```
MUL16_16:
       ;Multiply R5 by R7
       MOV A,R5      ;Move the R5 into the Accumulator
       MOV B,R7      ;Move R7 into B
       MUL AB        ;Multiply the two values
       MOV R2,B      ;Move B (the high-byte) into R2
       MOV R3,A      ;Move A (the low-byte) into R3
       ;Multiply R5 by R6
       MOV A,R5      ;Move R5 back into the Accumulator
       MOV B,R6      ;Move R6 into B
       MUL AB        ;Multiply the two values
       ADD A,R2      ;Add the low-byte into the value already in R2
       MOV R2,A      ;Move the resulting value back into R2
       MOV A,B       ;Move the high-byte into the accumulator
       ADDC A,#00h   ;Add zero (plus the carry, if any)
       MOV R1,A      ;Move the resulting answer into R1
       MOV A,#00h    ;Load the accumulator with  zero
       ADDC A,#00h   ;Add zero (plus the carry, if any)
       MOV R0,A      ;Move the resulting answer to R0.
       ;Multiply R4 by R7
       MOV A,R4      ;Move R4 into the Accumulator
       MOV B,R7      ;Move R7 into B
       MUL AB        ;Multiply the two values
       ADD A,R2      ;Add the low-byte into the value already in R2
       MOV R2,A      ;Move the resulting value back into R2
       MOV A,B       ;Move the high-byte into the accumulator
       ADDC A,R1     ;Add the current value of R1 (plus any carry)
       MOV R1,A      ;Move the resulting answer into R1.
       MOV A,#00h    ;Load the accumulator with zero
       ADDC A,R0     ;Add the current value of R0 (plus any carry)
       MOV R0,A      ;Move the resulting answer to R1.
       ;Multiply R4 by R6
       MOV A,R4      ;Move R4 back into the Accumulator
       MOV B,R6      ;Move R6 into B
       MUL AB        ;Multiply the two values
       ADD A,R1      ;Add the low-byte into the value already in R1
       MOV R1,A      ;Move the resulting value back into R1
       MOV A,B       ;Move the high-byte into the accumulator
       ADDC A,R0     ;Add it to the value already in R0 (plus any carry)
       MOV R0,A      ;Move the resulting answer back to R0
       ;Return - answer is now in R0, R1, R2, and R3
       RET
```

And to call our routine to multiply the two values we used in the example above, we'd use the code:

```
       ;Load the first value into R6 and R7
```

```
MOV R6,#62h
MOV R7,#30h

;Load the second value into R6 and R7
MOV R4,#43h
MOV R5,#2Eh

;Call the 16-bit subtraction routine
LCALL MUL16_16
```

12.5 16-bit Division

A frequent question asked by 8052 developers is how to do 16-bit division. At first glance division seems more complicated than addition, subtraction, or multiplication. However, upon analyzing the following approach, generously contributed by Jörg Rockstroh, we realize that 16-bit division isn't all that more complicated than the other three types of 16-bit math. 16-bit division utilizes bit shifting (RLC and RRC) and also 16-bit subtraction. With these capabilities it turns out that 16-bit division is surprisingly straightforward.

16-bit division is the division of one 16-bit value by another 16-bit value, returning a 16-bit quotient and a 16-bit remainder.

Programming Tip: The number of bits in the quotient and the remainder can never be larger than the number of bits in the original dividend. For example, if you are dividing a 16-bit value by a 2-bit value, both the quotient and the remainder must be able to handle a 16-bit result. If you are dividing a 24-bit value by a 16-bit value, the quotient and remainder must both be able to handle a 24-bit result.

First, as we did in the previous sections, let's recall how division is traditionally done in elementary school. In this case we will divide 179 by 8. These are actually two 8-bit numbers and the DIV opcode could be used; but we will use 179 by 8 for the sake of simplicity knowing that the same approach will work equally well with 16-bit values—it would just take longer to walk through the steps.

		2	2
8	**1**	**7**	**9**
	1	6	
		1	9
		1	6
			3

In this example we divide 179 by 8 by using the following steps:

1. We attempt to divide 8 into 1, but it doesn't fit.
2. We attempt to divide 8 into 17 and get 2 (first digit of the answer).
3. We multiply 8 by 2 (the result obtained in step #2) to get 16.
4. We subtract 16 from 17 and get 1.
5. We attempt to divide 8 by the 1 we got in step #4, but it doesn't fit.
6. We bring down the 9 from the dividend to arrive at the number 19.
7. We attempt to divide 8 into 19 and get 2 (second digit of the answer).
8. We subtract 16 from 19 (the number we divided into in step #7) and get 3.

9. The value "3" left over in step #8 is the "remainder" of the division, so the answer is 22 remainder 3.

This can be accomplished in code in the following manner:

1. Make the divisor the same length as the dividend. In our example the dividend was 3 digits long (179) so we need to make the divisor 3 digits long. Thus we add two zeros and end up with a divisor of 800.
2. 179 can't be divided by 800, so we "shift" the divisor to the right one digit. The 800 divisor becomes 80.
3. 179 can be divided by 80 twice, so the first digit of our answer is 2.
4. We multiply 2 (the result of step #3) by 80 (the current divisor) and get 160.
5. We subtract 160 (result of step #4) from the dividend and end up 19.
6. We "shift" the divisor to the right one digit. The 80 divisor becomes 8.
7. 19 (the result from step 5) can be divided by the divisor, 8, twice, so the next digit of our answer is 2.
8. We multiply 2 (the result of step #7) by 8 (the current divisor) and get 16.
9. We subtract 16 (the result of step #8) from the current dividend, 19, for an answer of 3.
10. There are no more digits to shift to the right so the "left-over" answer from step #9 is our remainder.

This same approach can be used for 16-bit binary division. Just keep in mind the following points:

1. Rather than determining the "largest" bit in the dividend and adjusting the divisor to that same bit position, we just shift the divisor as far to the left as we can. That is, 179 decimal is 0000000010110011 as a 16-bit binary value and 8 decimal is 0000000000001000. While we could check the and realize that the highest bit used in 179 is bit 7 and shift the divisor until its highest bit is also in bit 7, this turns out to be needlessly complicated. It is easier just to shift the value 8 all the way to the left so it becomes 1000000000000000.
2. When doing long division as described above it is necessary to determine at each step whether the divisor fits into the dividend. You are basically asking "Is divisor < dividend?" In binary all we do is attempt to do a 16-bit subtraction of divisor from dividend. If thjs produces a carry (carry bit set) then it means the divisor didn't fit into dividend and a "0" goes into that position of the answer. If the subtraction doesn't produce a carry (carry bit clear) then it means the divisor does fit into the dividend and a "1" goes into that position of the answer. Since this is binary math the answer is either 0 or 1.

We will now present the code used to accomplish each step of the 16-bit division.

Step 1: Shift Divisor as far left as possible

```
  MOV B,#00h ;Clear B.  B will count the number of left-shifted bits
Div1:
  INC B      ;Increment counter for each left shift, minimum 1 shift
  MOV A,R2   ;Move the current divisor low byte into the accumulator
  RLC A      ;Shift low-byte left, high bit goes to carry
  MOV R2,A   ;Save the updated divisor low-byte
  MOV A,R3   ;Move the current divisor high byte into the accumulator
  RLC A      ;Shift high-byte left, rotate carry from low-byte into low bit
  MOV R3,A   ;Save the updated divisor high-byte
  JNC div1   ;Repeat until carry flag is set from high-byte
```

In the case of our example, once the above code is executed the registers will be as follows (including the carry bit 'C'):

```
C/R1/R0 0 00000000 10110011
C/R3/R2 1 00000000 00000000
```

Step 2: Perform the actual division loop

At this point we can do the division itself. As we are in binary mode there is no need for a real division--it's just a comparison as described above. The comparison appears in the code as a 16-bit subtraction.

The first six instructions shifts the current 16-bit divisor one bit to the right. A safe (backup) copy of the dividend is then made in Internal RAM 06h and 07h. The next section of code subtracts the current shifted divisor from the dividend. If the subtraction occurred without error then it we know the that the division was successful and insert a "1" into the result byte at that position; otherwise we know the subtraction failed so we insert a "0" into the result byte at that position.

```
div2:        ;Perform division cycle
  ;Shift current divisor one bit to the right
  MOV A,R3   ;Move high-byte of divisor into accumulator
  RRC A      ;Rotate high-byte of divisor right and into carry
  MOV R3,A   ;Save updated value of high-byte of divisor
  MOV A,R2   ;Move low-byte of divisor into accumulator
  RRC A      ;Rotate low-byte of divisor right, with carry from high-byte
  MOV R2,A   ;Save updated value of low-byte of divisor
  CLR C      ;Clear carry, we don't need it anymore and must be clear for
             ;upcoming SUBB operations.

  ;Make a safe copy of the dividend so that if our subtraction
  ;shows that the divisor didn't fit intop the dividend we can
  ;restore the dividend to its current state.
  MOV 07h,R1 ;Make a safe copy of the dividend high-byte
  MOV 06h,R0 ;Make a safe copy of the dividend low-byte

  ;Subtract divisor from dividend, overwriting the current dividend
  ;This subtraction will tell us whether or not the divisor fits in
  ;the value of the dividend.
  MOV A,R0   ;Move low-byte of dividend into accumulator
  SUBB A,R2  ;Dividend - shifted divisor = result bit (no factor, only 0 or 1)
  MOV R0,A   ;Save updated dividend
  MOV A,R1   ;Move high-byte of dividend into accumulator
  SUBB A,R3  ;Subtract high-byte of divisor (all together 16-bit substraction)
  MOV R1,A   ;Save updated high-byte back in high-byte of divisor

  ;At this point if the carry flag is not set then it means that the
  ;subtraction occurred without a carry, meaning the divisor did fit in
  ;the dividend.  If the carry flag is set then it means the subtraction
  ;create a negative number, meaning the divisor did not fit in the
  ;dividend.
  JNC div3   ;If carry flag is NOT set, result is 1

  ;If we get here it means the carry bit was set.  This means the subtraction
  ;created a negative number which means the divisor didn't fit in the
  ;dividend.  Since we have overwritten the dividend with a now-invalid
  ;negative number, we restore the value of the dividend by restoring the
  ;safe values we stored prior to initiating the subtraction.
  MOV R1,07h ;Otherwise result is 0, save copy of divisor to undo subtraction
  MOV R0,06h
```

```
div3:
  ;At this point if the carry bit is set then it means the divisor didn't
  ;fit in the dividend, so the answer for this position in the result is “0”.
  ;If the carry bit is clear then the divisor did fit in the dividend, so the
  ;answer for this position in the result is “1”.  As you can see, the bit
  ;in the answer is the opposite of the carry bit; thus we invert the carry
  ;bit and insert it into the lowest bit of our result (R4 and R5).
  CPL C       ;Invert carry
  MOV A,R4    ;Get current low byte of the temporary result
  RLC A       ;Shift carry flag into low-bit of temporary result
  MOV R4,A    ;Update the low byte of the temporary result by shifting left
  MOV A,R5    ;Get current high byte of the temporary result
  RLC A       ;Shift high byte of the temporary result left
  MOV R5,A    ;Update the high byte of the temporary result
  DJNZ B,div2 ;Repeat division loop until "B" is zero
```

Step 3. Final Clean-up.

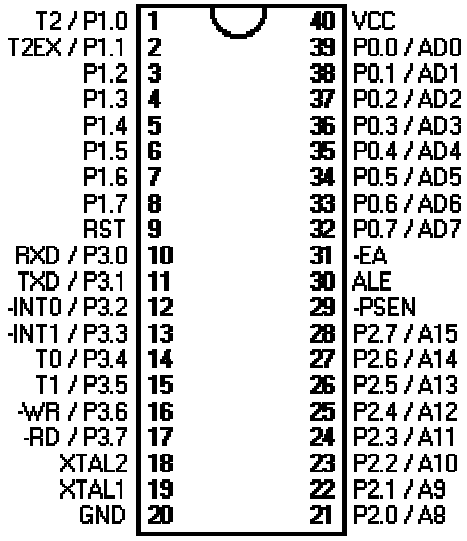
Since our working result is stored in R4 and R5, we finish the routine by moving the answer into R2 and R3.

```
  MOV R3,05h    ;Move result to R3/R2
  MOV R2,04h    ;Move result to R3/R2
```

To use the routine, call it with R0/R1 with the low/hi byte of the dividend, R2/R3 with the low/hi byte of the divisor. The routine leaves the quotient in R2 and R3 and leaves the remainder in R0 and R1.

Chapter 13: 8052 Microcontroller Pin Functions

While there are many packages in which 8052 and derivatives are produced, including surface-mount, etc. this chapter will explain the traditional 8052 40-pin DIP pinout. The names and functions of these pins are almost always found on 8052-compatible parts although they are certainly found at different pin numbers. In all cases you should consult the part's datasheet to confirm the function of each pin.



13.1 I/O Ports (P0, P1, P2, P3)

Of the 40 pins of the typical 8052, 32 of them are dedicated to I/O lines that have a one-to-one relation with SFRs P0, P1, P2, and P3. The developer may raise and lower these lines by writing 1s or 0s to the corresponding bits in the SFRs. Likewise, the current state of these lines may be read by reading the corresponding bits of the SFRs.

All of the ports have internal pull-up resistors except for port 0.

13.1.1 Port 0

Port 0 is dual-function in that it in some designs port 0's I/O lines are available to the developer to access external devices while in other designs it is used to access external memory. If the circuit requires external RAM or ROM, the microcontroller will automatically use port 0 to clock in/out the 8-bit data word as well as the low 8 bits of the address in response to a MOVX instruction and port 0 I/O lines may be used for other functions as long as external RAM isn't being accessed at the same time. If the circuit requires external code memory, the microcontroller will automatically use the port 0 I/O lines to access each instruction that is to be executed. In this case, port 0 cannot be utilized for other purposes since the state of the I/O lines are constantly being modified to access external code memory.

Note that there are no pull-up resistors on port 0, so it may be necessary to include your own pull-up resistors depending on the characteristics of the parts you will be driving via port 0.

13.1.2 Port 1

Port 1 consists of 8 I/O lines that you may use exclusively to interface to external parts. Unlike port 0, typical derivatives do not use port 1 for any functions themselves. Port 1 is commonly used to interface to external hardware such as LCDs, keypads, and other devices. With 8052 derivatives, two bits of port 1 are optionally used as described for extended timer 2 functions. These two lines are not assigned these special functions on 8051's since 8051's don't have a timer 2. Further, these lines can still be used for your own purposes if you don't need these features of timer 2.

P1.0 (T2): If T2CON.1 is set (C/T2), then timer 2 will be incremented whenever there is a 1-0 transition on this line. With C/T2 set, P1.0 is the clock source for timer 2.

P1.1 (T2EX): If timer 2 is in auto-reload mode and T2CON.3 (EXEN2) is set, a 1-0 transition on this line will cause timer 2 to be reloaded with the auto-reload value. This will also cause the T2CON.6 (EXF2) external flag to be set, which may cause an interrupt if so enabled.

13.1.3 Port 2

Like port 0, port 2 is dual-function. In some circuit designs it is available for accessing devices while in others it is used to address external RAM or external code memory. When the MOVX @DPTR instruction is used, port 2 is used to output the high byte of the memory address that is to be accessed. In these cases, port 2 may be used to access other devices as long as the devices are not being accessed at the same time a MOVX instruction is using port 2 to address external RAM. If the circuit requires external code memory, the microcontroller will automatically use the port 2 I/O lines to access each instruction that is to be executed. In this case, port 2 cannot be utilized for other purposes since the state of the I/O lines are constantly being modified to access external code memory.

Tip: Port 2 is only modified automtically by the microcontroller when accessing external code memory or when the MOVX @DPTR instruction is used. Port 2 is not affected by the MOVX @Ri instructions. It is, of course, affected by instructions that specifically address it, such as MOV P2,#20h.

13.1.4 Port 3

Port 3 consists entirely of dual-function I/O lines. While the developer may access all these lines from their software by reading/writing to the P3 SFR, each pin has a pre-defined function that the microcontroller handles automatically when configured to do so and/or when necessary.

P3.0 (RXD): The UART/serial port uses P3.0 as the "receive line." In circuit designs that will be using the microcontroller's internal serial port, this is the line into which serial data will be clocked. Note that when interfacing an 8052 to an RS-232 port that you may not connect this line directly to the RS-232 pin; rather, you must pass it through a part such as the MAX232 to obtain the correct voltage levels. This pin is available for any use the developer may assign it if the circuit has no need to receive data via the integrated serial port.

P3.1 (TXD): The UART/serial port uses P3.1 as the "transmit line." In circuit designs that will be using the microcontroller's internal serial port, this is the line that the microcontroller will clock out all data which is written to the SBUF SFR. Note that when interfacing an 8052 to an RS-232 port that you may not connect this line directly to the RS-232 pin; rather, you must pass it through a part such as the

MAX232 to obtain the correct voltage levels. This pin is available for any use the developer may assign it if the circuit has no need to transmit data via the integrated serial port.

P3.2 (-INT0): When so configured, this line is used to trigger an "External 0 Interrupt." This may either be low-level triggered or may be triggered on a 1-0 transition. Please see the chapter on interrupts for details. This pin is available for any use the developer may assign it if the circuit does not need to trigger an external 0 interrupt.

P3.3 (-INT1): When so configured, this line is used to trigger an "External 1 Interrupt." This may either be low-level triggered or may be triggered on a 1-0 transition. Please see the chapter on interrupts for details. This pin is available for any use the developer may assign it if the circuit does not need to trigger an external 1 interrupt.

P3.4 (T0): When so configured, this line is used as the clock source for timer 0. Timer 0 will be incremented either every instruction cycle that T0 is high or every time there is a 1-0 transition on this line, depending on how the timer is configured. Please see the chapter on timers for details. This pin is available for any use the developer may assign it if the circuit does not to control timer 0 externally.

P3.5 (T1): When so configured, this line is used as the clock source for timer 1. Timer 1 will be incremented either every instruction cycle that T1 is high or every time there is a 1-0 transition on this line, depending on how the timer is configured. Please see the chapter on timers for details. This pin is available for any use the developer may assign it if the circuit does not to control timer 1 externally.

P3.6 (-WR): This is external memory write strobe line. This line will be asserted low by the microcontroller whenever a MOVX instruction writes to external RAM. This line should be connected to the RAM's write (-W) line. This pin is available for any use the developer may assign it if the circuit does not write to external RAM using MOVX.

P3.7 (-RD): This is external memory write strobe line. This line will be asserted low by the microcontroller whenever a MOVX instruction writes to external RAM. This line should be connected to the RAM's write (-W) line. This pin is available for any use the developer may assign it if the circuit does not read from external RAM using MOVX.

13.2 Oscillator Inputs (XTAL1, XTAL2)

The 8052 is typically driven by a crystal connected to pins 18 (XTAL2) and 19 (XTAL1). Common crystal frequencies are 11.0592Mhz as well as 12Mhz, although many newer derivatives are capable of accepting frequencies as high as 40Mhz.

While a crystal is the normal clock source, this isn't necessarily the case. A TTL clock source may also be attached to XTAL1 and XTAL2 to provide the microcontroller's clock.

13.3 Reset Line (RST)

Pin 9 is the master reset line for the microcontroller. When this pin is brought high for two instruction cycles, the microcontroller is effectively reset. SFRs, including the I/O ports, are restored to their default conditions and the program counter will be reset to 0000h. Keep in mind that Internal RAM is *not* affected by a reset. The microcontroller will begin executing code at 0000h when pin 9 returns to a low state.

The reset line is often connected to a reset button/switch that the user may press to reset the circuit. It is also common to connect the reset line to a watchdog IC or a supervisor IC (such as MAX707). The latter is highly recommended for commercial and professional designs since traditional resistor-capacitor networks attached to the reset line, while often sufficient for students or hobbyists, are not terribly reliable.

13.4 Address Latch Enable (ALE)

The ALE at pin 30 is an output-only pin that is controlled entirely by the microcontroller and allows the microcontroller to multiplex the low-byte of a memory address and the 8-bit data itself on port 0. This is because, while the high-byte of the memory address is sent on port 2, port 0 is used both to send the low-byte of the memory address and the data itself. This is accomplished by placing the low-byte of the address on port 0, exerting ALE high to latch the low-byte of the address into a latch IC (such as the 74HC573), and then placing the 8 data-bits on port 0. In this way the 8052 is able to output a 16-bit address and an 8-bit data word with 16 I/O lines instead of 24.

The ALE line is used in this fashion both for accessing external RAM with MOVX @DPTR as well as for accessing instructions in external code memory. When your program is executed from external code memory, ALE will pulse at a rate of $1/6^{th}$ that of the oscillator frequency. Thus if the oscillator is operating at 11.0592Mhz, ALE will pulse at a rate of 1,843,200 times per second. The only exception is when the MOVX instruction is executed one ALE pulse is missed in lieu of a pulse on –WR or –RD.

13.5 Program Store Enable (-PSEN)

The Program Store Enable (PSEN) line at pin 29 is exerted low automatically by the microcontroller whenever it accesses external code memory. This line should be attached to the Output Enable (-OE) pin of the EPROM that contains your code memory.

PSEN will not be exerted by the microcontroller and will remain in a high state if your program is being executed from internal code memory.

13.6 External Access (-EA)

The External Access (-EA) line at pin 31 is used to determine whether the 8052 will execute your program from external code memory or from internal code memory. If EA is tied high (connected to +5V) then the microcontroller will execute the program it finds in internal/on-chip code memory. If EA is tied low (to ground) then it will attempt to execute the program it finds in the attached external code memory EPROM. Of course, your EPROM must be properly connected for the microcontroller to be able to access your program in external code memory.

EA must be tied low for any microcontroller that does not have internal code memory.

Tip: Even when EA is tied high, indicating that the microcontroller should execute from internal code memory, the microcontroller will normally attempt to execute from external code memory if the program counter references an address not available for that chip. For example, if the derivative you are using as 4k of internal code-memory and you tie EA high, the derivative will start executing the program it finds on-chip. However, if your on-chip program attempts to execute code above 0FFFh (i.e. exceeding 4k) then the derivative will normally attempt to execute that code at that address from external code memory. Thus it is possible to have a "split" design where some of your code is found on-chip and the rest is found off-chip. This can potentially be very useful from a security standpoint where your proprietary, trade secret code

could be programmed in the microcontroller itself and protected with security bits, while less sensitive code could be stored on an external EPROM. You could also program the microcontroller with underlying "solid" firmware that will never change and use a socketed EPROM to carry sections of code that may be subject to upgrades or changes in the future.

Appendix A: 8052 Instruction Set Quick-Reference

00	NOP	40	JC *relAddr*	80	SJMP *relAddr*	C0	PUSH *direct*
01	AJMP *pg0Addr*	41	AJMP *pg2Addr*	81	AJMP *pg4Addr*	C1	AJMP *pg6Addr*
02	LJMP *addr16*	42	ORL *direct*,A	82	ANL C,*bitAddr*	C2	CLR *bitAddr*
03	RR A	43	ORL *direct*,*#data8*	83	MOVC A,@A+PC	C3	CLR C
04	INC A	44	ORL A,*#data8*	84	DIV AB	C4	SWAP A
05	INC *direct*	45	ORL A,*direct*	85	MOV *direct,direct*	C5	XCH A,*direct*
06	INC @R0	46	ORL A,@R0	86	MOV *direct*,@R0	C6	XCH A,@R0
07	INC @R1	47	ORL A,@R1	87	MOV *direct*,@R1	C7	XCH A,@R1
08	INC R0	48	ORL A,R0	88	MOV *direct*,R0	C8	XCH A,R0
09	INC R1	49	ORL A,R1	89	MOV *direct*,R1	C9	XCH A,R1
0A	INC R2	4A	ORL A,R2	8A	MOV *direct*,R2	CA	XCH A,R2
0B	INC R3	4B	ORL A,R3	8B	MOV *direct*,R3	CB	XCH A,R3
0C	INC R4	4C	ORL A,R4	8C	MOV *direct*,R4	CC	XCH A,R4
0D	INC R5	4D	ORL A,R5	8D	MOV *direct*,R5	CD	XCH A,R5
0E	INC R6	4E	ORL A,R6	8E	MOV *direct*,R6	CE	XCH A,R6
0F	INC R7	4F	ORL A,R7	8F	MOV *direct*,R7	CF	XCH A,R7
10	JBC *bitAddr,relAddr*	50	JNC *relAddr*	90	MOV DPTR,*#data16*	D0	POP *direct*
11	ACALL *pg0Addr*	51	ACALL *pg2Addr*	91	ACALL *pg4Addr*	D1	ACALL *pg5Addr*
12	LCALL *address16*	52	ANL *direct*,A	92	MOV *bitAddr*,C	D2	SETB *bitAddr*
13	RRC A	53	ORL *direct*,*#data8*	93	MOVC A,@DPTR	D3	SETB C
14	DEC A	54	ANL A,*#data8*	94	SUBB A,*#data8*	D4	DA A
15	DEC *direct*	55	ANL A,*direct*	95	SUBB A,*direct*	D5	DJNZ *direct,relAddr*
16	DEC @R0	56	ANL A,@R0	96	SUBB A,@R0	D6	XCHD A,@R0
17	DEC @R1	57	ANL A,@R1	97	SUBB A,@R1	D7	XCHD A,@R1
18	DEC R0	58	ANL A,R0	98	SUBB A,R0	D8	XCHD A,R0
19	DEC R1	59	ANL A,R1	99	SUBB A,R1	D9	XCHD A,R1
1A	DEC R2	5A	ANL A,R2	9A	SUBB A,R2	DA	XCHD A,R2
1B	DEC R3	5B	ANL A,R3	9B	SUBB A,R3	DB	XCHD A,R3
1C	DEC R4	5C	ANL A,R4	9C	SUBB A,R4	DC	XCHD A,R4
1D	DEC R5	5D	ANL A,R5	9D	SUBB A,R5	DD	XCHD A,R5
1E	DEC R6	5E	ANL A,R6	9E	SUBB A,R6	DE	XCHD A,R6
1F	DEC R7	5F	ANL A,R7	9F	SUBB A,R7	DF	XCHD A,R7
20	JB *bitAddr,relAddr*	60	JZ *relAddr*	A0	ORL C,*/bitAddr*	E0	MOVX A,@DPTR
21	AJMP *pg1Addr*	61	AJMP *pg3Addr*	A1	AJMP *pg5Addr*	E1	AJMP *pg7Addr*
22	RET	62	XRL *direct*,A	A2	MOV C,*bitAddr*	E2	MOVX A,@R0
23	RL A	63	XRL *direct*,*#data8*	A3	INC DPTR	E3	MOVX A,@R1
24	ADD A,*#data8*	64	XRL A,*#data8*	A4	MUL AB	E4	CLR A
25	ADD A,*direct*	65	XRL A,*direct*	A5		E5	MOV A,*direct*
26	ADD A,@R0	66	XRL A,@R0	A6	MOV @R0,*direct*	E6	MOV A,@R0
27	ADD A,@R1	67	XRL A,@R1	A7	MOV @R1,*direct*	E7	MOV A,@R1
28	ADD A,R0	68	XRL A,R0	A8	MOV R0,*direct*	E8	MOV A,R0
29	ADD A,R1	69	XRL A,R1	A9	MOV R1,*direct*	E9	MOV A,R1
2A	ADD A,R2	6A	XRL A,R2	AA	MOV R2,*direct*	EA	MOV A,R2
2B	ADD A,R3	6B	XRL A,R3	AB	MOV R3,*direct*	EB	MOV A,R3
2C	ADD A,R4	6C	XRL A,R4	AC	MOV R4,*direct*	EC	MOV A,R4
2D	ADD A,R5	6D	XRL A,R5	AD	MOV R5,*direct*	ED	MOV A,R5
2E	ADD A,R6	6E	XRL A,R6	AE	MOV R6,*direct*	EE	MOV A,R6
2F	ADD A,R7	6F	XRL A,R7	AF	MOV R7,*direct*	EF	MOV A,R7
30	JNB *bitAddr,relAddr*	70	JNZ *relAddr*	B0	ANL C,*/bitAddr*	F0	MOVX @DPTR,A
31	ACALL *pg1Addr*	71	ACALL *pg3Addr*	B1	ACALL *pg5Addr*	F1	ACALL *pg7Addr*
32	RETI	72	ORL C,*bitAddr*	B2	CPL *bitAddr*	F2	MOVX @R0,A
33	RLC A	73	JMP @A+DPTR	B3	CPL C	F3	MOVX @R1,A
34	ADDC A,#data	74	MOV A,*#data8*	B4	CJNE A,*#data8,relAddr*	F4	CPL A
35	ADDC A,*direct*	75	MOV *direct*,*#data8*	B5	CJNE A,*direct,relAddr*	F5	MOV *direct*,A
36	ADDC A,@R0	76	MOV @R0,*#data8*	B6	CJNE @R0,*#data8,relAddr*	F6	MOV @R0,A
37	ADDC A,@R1	77	MOV @R1,*#data8*	B7	CJNE @R1,*#data8,relAddr*	F7	MOV @R1,A
38	ADDC A,R0	78	MOV R0,*#data8*	B8	CJNE R0,*#data8,relAddr*	F8	MOV R0,A
39	ADDC A,R1	79	MOV R1,*#data8*	B9	CJNE R1,*#data8,relAddr*	F9	MOV R1,A
3A	ADDC A,R2	7A	MOV R2,*#data8*	BA	CJNE R2,*#data8,relAddr*	FA	MOV R2,A
3B	ADDC A,R3	7B	MOV R3,*#data8*	BB	CJNE R3,*#data8,relAddr*	FB	MOV R3,A
3C	ADDC A,R4	7C	MOV R4,*#data8*	BC	CJNE R4,*#data8,relAddr*	FC	MOV R4,A
3D	ADDC A,R5	7D	MOV R5,*#data8*	BD	CJNE R5,*#data8,relAddr*	FD	MOV R5,A
3E	ADDC A,R6	7E	MOV R6,*#data8*	BE	CJNE R6,*#data8,relAddr*	FE	MOV R6,A
3F	ADDC A,R7	7F	MOV R7,*#data8*	BF	CJNE R7,*#data8,relAddr*	FF	MOV R7,A

Appendix B: 8052 Instruction Set

This appendix is a reference for all instructions in the 8052 instruction set. For each instruction, the following information is provided:

- **Instruction**: Indicates the correct syntax for the given opcode.
- **OpCode**: The operation code, in the range of 0x00 through 0xFF, that represents the given instruction in machine code.
- **Bytes**: The total number of bytes (including the opcode byte) that make up the instruction.
- **Cycles**: The number of machine cycles required to execute the instruction.
- **Flags**: The flags that are modified by the instruction, if any.

When listing instruction syntax, the following terms will be used:

- *bitAddr*: Bit address value (00-FF)
- *pgXAddr*: Absolute 2k (13-bit) Address
- *data8*: Immediate 8-bit data value
- *data16*: Immedate 16-bit data value
- *address16*: 16-bit code address
- *direct*: Direct address (IRAM 00-7F, SFR 80-FF)
- *relAddr*: Relative address (-127 to +128 bytes)

ACALL – Absolute Call within 2k Block

Syntax: ACALL *codeAddress*

Instructions	OpCode	Bytes	Cycles	Flags
ACALL *pg0Addr*	0x11	2	2	None
ACALL *pg1Addr*	0x31	2	2	None
ACALL *pg2Addr*	0x51	2	2	None
ACALL *pg3Addr*	0x71	2	2	None
ACALL *pg4Addr*	0x91	2	2	None
ACALL *pg5Addr*	0xB1	2	2	None
ACALL *pg6Addr*	0xD1	2	2	None
ACALL *pg7Addr*	0xF1	2	2	None

ACALL unconditionally calls a subroutine at the indicated code address. ACALL pushes the address of the instruction that follows ACALL onto the stack, least-significant-byte first, most-significant-byte second. The Program Counter is then updated so that program execution continues at the indicated address.

The new value for the Program Counter is calculated by replacing the least-significant-byte of the Program Counter with the second byte of the ACALL instruction, and replacing bits 0-2 of the most-significant-byte of the Program Counter with bits 5-7 of the opcode value. Bits 3-7 of the most-significant-byte of the Program Counter remain unchaged.

Since only 11 bits of the Program Counter are affected by ACALL, calls may only be made to routines located within the same 2k block as the first byte that follows ACALL.

See Also: LCALL, RET

ADD, ADDC – Add Value, Add Value with Carry

Syntax: ADD A,*operand*
Syntax: ADDC A,*operand*

Instructions	OpCode	Bytes	Cycles	Flags
ADD A,#*data8*	0x24	2	1	C, AC, OV
ADD A,*direct*	0x25	2	1	C, AC, OV
ADD A,@R0	0x26	1	1	C, AC, OV
ADD A,@R1	0x27	1	1	C, AC, OV
ADD A,R0	0x28	1	1	C, AC, OV
ADD A,R1	0x29	1	1	C, AC, OV
ADD A,R2	0x2A	1	1	C, AC, OV
ADD A,R3	0x2B	1	1	C, AC, OV
ADD A,R4	0x2C	1	1	C, AC, OV
ADD A,R5	0x2D	1	1	C, AC, OV
ADD A,R6	0x2E	1	1	C, AC, OV
ADD A,R7	0x2F	1	1	C, AC, OV
ADDC A,#*data8*	0x34	2	1	C, AC, OV
ADDC A,*direct*	0x35	2	1	C, AC, OV
ADDC A,@R0	0x36	1	1	C, AC, OV
ADDC A,@R1	0x37	1	1	C, AC, OV
ADDC A,R0	0x38	1	1	C, AC, OV
ADDC A,R1	0x39	1	1	C, AC, OV
ADDC A,R2	0x3A	1	1	C, AC, OV
ADDC A,R3	0x3B	1	1	C, AC, OV
ADDC A,R4	0x3C	1	1	C, AC, OV
ADDC A,R5	0x3D	1	1	C, AC, OV
ADDC A,R6	0x3E	1	1	C, AC, OV
ADDC A,R7	0x3F	1	1	C, AC, OV

ADD and ADDC both add the value operand to the value of the Accumulator, leaving the resulting value in the Accumulator. The value operand is not affected. ADD and ADDC function identically except that ADDC adds the value of operand as well as the value of the Carry flag whereas ADD does not add the Carry flag to the result.

The **Carry bit** (**C**) is set if there is a carry-out of bit 7. In other words, if the unsigned summed value of the Accumulator, operand and (in the case of ADDC) the Carry flag exceeds 255 Carry is set. Otherwise, the Carry bit is cleared.

The **Auxillary Carry** (**AC**) bit is set if there is a carry-out of bit 3. In other words, if the unsigned summed value of the low nibble of the Accumulator, operand and (in the case of ADDC) the Carry flag exceeds 15 the Auxillary Carry flag is set. Otherwise, the Auxillary Carry flag is cleared.

The **Overflow** (**OV**) bit is set if there is a carry-out of bit 6 or out of bit 7, but not both. In other words, if the addition of the Accumulator, operand and (in the case of ADDC) the Carry flag treated as signed values results in a value that is out of the range of a signed byte (-128 through +127) the Overflow flag is set. Otherwise, the Overflow flag is cleared.

See Also: SUBB, DA, INC, DEC

AJMP – Absolute Jump within 2k Block

Syntax: AJMP *codeAddress*

Instructions	OpCode	Bytes	Cycles	Flags
AJMP *pg0Addr*	0x01	2	2	None
AJMP *pg1Addr*	0x21	2	2	None
AJMP *pg2Addr*	0x41	2	2	None
AJMP *pg3Addr*	0x61	2	2	None
AJMP *pg4Addr*	0x81	2	2	None
AJMP *pg5Addr*	0xA1	2	2	None
AJMP *pg6Addr*	0xC1	2	2	None
AJMP *pg7Addr*	0xE1	2	2	None

AJMP unconditionally jumps to the indicated *codeAddress*. The new value for the Program Counter is calculated by replacing the least-significant-byte of the Program Counter with the second byte of the AJMP instruction, and replacing bits 0-2 of the most-significant-byte of the Program Counter with bits 5-7 of the opcode value. Bits 3-7 of the most-significant-byte of the Program Counter remain unchaged.

Since only 11 bits of the Program Counter are affected by AJMP, jumps may only be made to code located within the same 2k block as the first byte that follows AJMP.

See Also: LJMP, SJMP

ANL– Bitwise AND

Syntax: ANL *operand1*,*operand2*

Instructions	OpCode	Bytes	Cycles	Flags
ANL *direct*,A	0x52	2	1	None
ANL *direct*,#*data8*	0x53	3	2	None
ANL A,#*data8*	0x54	2	1	None
ANL A,*direct*	0x55	2	1	None
ANL A,@R0	0x56	1	1	None
ANL A,@R1	0x57	1	1	None
ANL A,R0	0x58	1	1	None
ANL A,R1	0x59	1	1	None
ANL A,R2	0x5A	1	1	None
ANL A,R3	0x5B	1	1	None
ANL A,R4	0x5C	1	1	None
ANL A,R5	0x5D	1	1	None
ANL A,R6	0x5E	1	1	None
ANL A,R7	0x5F	1	1	None
ANL C,*bitAddr*	0x82	2	1	C
ANL C,/*bitAddr*	0xB0	2	1	C

ANL does a bitwise "AND" operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of *operand2* is not affected. A logical "AND" compares the bits of each operand and sets the corresponding bit in the resulting byte only if the bit was set in both of the original operands, otherwise the resulting bit is cleared.

See Also: ORL, XRL

CJNE – Compare and Jump if Not Equal

Syntax: CJNE *operand1*,*operand2*,*reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
CJNE A,#data8,reladdr	0xB4	3	2	C
CJNE A,*direct*,*reladdr*	0xB5	3	2	C
CJNE @R0,#*data8*,*reladdr*	0xB6	3	2	C
CJNE @R1,#*data8*,*reladdr*	0xB7	3	2	C
CJNE R0,#*data8*,*reladdr*	0xB8	3	2	C
CJNE R1,#*data8*,*reladdr*	0xB9	3	2	C
CJNE R2,#*data8*,*reladdr*	0xBA	3	2	C
CJNE R3,#*data8*,*reladdr*	0xBB	3	2	C
CJNE R4,#*data8*,*reladdr*	0xBC	3	2	C
CJNE R5,#*data8*,*reladdr*	0xBD	3	2	C
CJNE R6,#*data8*,*reladdr*	0xBE	3	2	C
CJNE R7,#*data8*,*reladdr*	0xBF	3	2	C

CJNE compares the value of *operand1* and *operand2* and branches to the indicated relative address if the two operands are not equal. If the two operands are equal program flow continues with the instruction following the CJNE instruction.

The **Carry bit** (**C**) is set if *operand1* is less than *operand2*, otherwise it is cleared.

See Also: DJNZ

CLR – Clear Register

Syntax: CLR *register*

Instructions	OpCode	Bytes	Cycles	Flags
CLR *bitAddr*	0xC2	2	1	None
CLR C	0xC3	1	1	C
CLR A	0xE4	1	1	None

CLR clears (sets to 0) all the bit(s) of the indicated register. If the register is a bit (including the carry bit), only the specified bit is affected. Clearing the Accumulator sets the Accumulator's value to 0.

See Also: SETB

CPL – Complement Register

Syntax: CPL *operand*

Instructions	OpCode	Bytes	Cycles	Flags
CPL A	0xF4	1	1	None
CPL C	0xB3	1	1	C
CPL *bitAddr*	0xB2	2	1	None

CPL complements *operand*, leaving the result in *operand*. If *operand* is a single bit then the state of the bit will be reversed. If *operand* is the Accumulator then all the bits in the Accumulator will be reversed. This can be thought of as "Accumulator Logical Exclusive OR 255" or as "255-Accumulator." If *operand* refers to a bit of an output port, the value that will be complemented is based on the last value written to that bit, not the last value read from it.

See Also: CLR, SETB

DA– Decimal Adjust Accumulator

Syntax: DA A

Instructions	OpCode	Bytes	Cycles	Flags
DA A	0xD4	1	1	C

DA adjusts the contents of the Accumulator to correspond to a BCD (Binary Coded Decimal) number after two BCD numbers have been added by the ADD or ADDC instruction.

If the carry bit is set or if the value of bits 0-3 exceed 9, 0x06 is added to the accumulator. If the carry bit was set when the instruction began, or if 0x06 was added to the accumulator in the first step, 0x60 is added to the accumulator.

The **Carry bit** (**C**) is set if the resulting value is greater than 0x99, otherwise it is cleared.

See Also: ADD, ADDC

DEC – Decrement Register

Syntax: DEC *register*

Instructions	OpCode	Bytes	Cycles	Flags
DEC A	0x14	1	1	None
DEC *direct*	0x15	2	1	None
DEC @R0	0x16	1	1	None
DEC @R1	0x17	1	1	None
DEC R0	0x18	1	1	None
DEC R1	0x19	1	1	None
DEC R2	0x1A	1	1	None
DEC R3	0x1B	1	1	None
DEC R4	0x1C	1	1	None
DEC R5	0x1D	1	1	None
DEC R6	0x1E	1	1	None
DEC R7	0x1F	1	1	None

DEC decrements the value of *register* by 1. If the initial value of *register* is 0, decrementing the value will cause it to reset to 255 (0xFF Hex). Note: The Carry Flag is *not* set when the value "rolls over" from 0 to 255.

See Also: INC, SUBB

DIV – Divide Accumulator by B

Syntax: DIV AB

Instructions	OpCode	Bytes	Cycles	Flags
DIV AB	0x84	1	1	C, OV

Divides the unsigned value of the Accumulator by the unsigned value of the "B" register. The resulting quotient is placed in the Accumulator and the remainder is placed in the "B" register.

The **Carry flag** (**C**) is always cleared.

The **Overflow flag** (**OV**) is set if division by 0 was attempted, otherwise it is cleared.

See Also: MUL AB

DJNZ – Decrement and Jump if Not Zero

Syntax: DJNZ *register,relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
DJNZ *direct,relAddr*	0xD5	3	2	None
DJNZ R0,*relAddr*	0xD8	2	2	None
DJNZ R1,*relAddr*	0xD9	2	2	None
DJNZ R2,*relAddr*	0xDA	2	2	None
DJNZ R3,*relAddr*	0xDB	2	2	None
DJNZ R4,*relAddr*	0xDC	2	2	None
DJNZ R5,*relAddr*	0xDD	2	2	None
DJNZ R6,*relAddr*	0xDE	2	2	None
DJNZ R7,*relAddr*	0xDF	2	2	None

DJNZ decrements the value of *register* by 1. If the initial value of *register* is 0, decrementing the value will cause it to reset to 255 (0xFF Hex). If the new value of register is *not* 0 the program will branch to the address indicated by *relAddr*. If the new value of *register* is 0, program flow continues with the instruction following the DJNZ instruction.

See Also: DEC, JZ, JNZ

INC – Increment Reister

Syntax: INC *register*

Instructions	OpCode	Bytes	Cycles	Flags
INC A	0x04	1	1	None
INC *direct*	0x05	2	1	None
INC @R0	0x06	1	1	None
INC @R1	0x07	1	1	None
INC R0	0x08	1	1	None
INC R1	0x09	1	1	None
INC R2	0x0A	1	1	None
INC R3	0x0B	1	1	None
INC R4	0x0C	1	1	None
INC R5	0x0D	1	1	None
INC R6	0x0E	1	1	None
INC R7	0x0F	1	1	None
INC DPTR	0xA3	1	2	None

INC increments the value of *register* by 1. If the initial value of register is 255 (0xFF Hex), incrementing the value will cause it to reset to 0. Note: The Carry Flag is *not* set when the value "rolls over" from 255 to 0.

In the case of "INC DPTR", the two-byte value of DPTR is incremented as an unsigned integer. If the initial value of DPTR is 65535 (0xFFFF Hex), incrementing the value will cause it to reset to 0. Again, the Carry Flag is *not* set when the value of DPTR "rolls over" from 65535 to 0.

See Also: ADD, ADDC, DEC

JB – Jump if Bit Set

Syntax: JB *bitAddr,relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
JB *bitAddr,relAddr*	0x20	3	2	None

JB branches to the address indicated by *relAddr* if the bit indicated by *bitAddr* is set. If the bit is not set program execution continues with the instruction following the JB instruction.

See Also: JBC, JNB

JBC – Jump if Bit Set and Clear Bit

Syntax: JBC *bitAddr,relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
JBC *bitAddr,reladdr*	0x10	3	2	None

JBC will branch to the address indicated by *relAddr* if the bit indicated by *bitAddr* is set. Before branching to *relAddr* the instruction will clear the indicated bit. If the bit is not set program execution continues with the instruction following the JBC instruction and the value of the bit is not changed.

See Also: JB, JNB

JC – Jump if Carry Set

Syntax: JC *relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
JC *relAddr*	0x40	2	2	None

JC will branch to the address indicated by *relAddr* if the Carry Bit is set. If the Carry Bit is not set program execution continues with the instruction following the JC instruction.

See Also: JNC

JMP – Jump to Data Pointer + Accumulator

Syntax: JMP @A+DPTR

Instructions	OpCode	Bytes	Cycles	Flags
JMP @A+DPTR	0x73	1	2	None

JMP jumps unconditionally to the address represented by the sum of the value of DPTR and the value of the Accumulator.

See Also: LJMP, AJMP, SJMP

JNB – Jump if Bit Not Set

Syntax: JNB *bitAddr,reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JNB *bitAddr,relAddr*	0x30	3	2	None

JNB will branch to the address indicated by *relAddr* if the indicated bit is not set. If the bit is set program execution continues with the instruction following the JNB instruction.

See Also: JB, JBC

JNC – Jump if Carry Not Set

Syntax: JNC *reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JNC *relAddr*	0x50	2	2	None

JNC branches to the address indicated by *relAddr* if the carry bit is not set. If the carry bit is set program execution continues with the instruction following the JNB instruction.

See Also: JC

JNZ – Jump if Accumulator Not Zero

Syntax: JNZ *reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JNZ *relAddr*	0x70	2	2	None

JNZ will branch to the address indicated by *relAddr* if the Accumulator contains any value except 0. If the value of the Accumulator is zero program execution continues with the instruction following the JNZ instruction.

See Also: JZ

JZ – Jump if Accumulator Zero

Syntax: JZ *reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JZ *relAddr*	0x60	2	2	None

JZ branches to the address indicated by *relAddr* if the Accumulator contains the value 0. If the value of the Accumulator is non-zero program execution continues with the instruction following the JNZ instruction.

See Also: JNZ

LCALL – Long Call

Syntax: LCALL *address16*

Instructions	OpCode	Bytes	Cycles	Flags
LCALL *address16*	0x12	3	2	None

LCALL calls a program subroutine. LCALL increments the program counter by 3 (to point to the instruction following LCALL) and pushes that value onto the stack , low-byte first, high-byte second. The Program Counter is then set to the 16-bit value *address16*, causing program execution to continue at that address.

See Also: ACALL, RET

LJMP – Long Jump

Syntax: LJMP *address16*

Instructions	OpCode	Bytes	Cycles	Flags
LJMP *address16*	0x02	3	2	None

LJMP jumps unconditionally to the specified *address16*.

See Also: AJMP, SJMP, JMP

MOV – Move Memory into/out of Accumulator

Syntax: MOV *operand1*, *operand2*

Instructions	OpCode	Bytes	Cycles	Flags
MOV A,#*data8*	0x74	2	1	None
MOV A,@R0	0xE6	1	1	None
MOV A,@R1	0xE7	1	1	None
MOV @R0,A	0xF6	1	1	None
MOV @R1,A	0xF7	1	1	None
MOV A,R0	0xE8	1	1	None
MOV A,R1	0xE9	1	1	None
MOV A,R2	0xEA	1	1	None
MOV A,R3	0xEB	1	1	None
MOV A,R4	0xEC	1	1	None
MOV A,R5	0xED	1	1	None
MOV A,R6	0xEE	1	1	None
MOV A,R7	0xEF	1	1	None
MOV A,*direct*	0xE5	2	1	None
MOV R0,A	0xF8	1	1	None
MOV R1,A	0xF9	1	1	None
MOV R2,A	0xFA	1	1	None
MOV R3,A	0xFB	1	1	None
MOV R4,A	0xFC	1	1	None
MOV R5,A	0xFD	1	1	None
MOV R6,A	0xFE	1	1	None
MOV R7,A	0xFF	1	1	None
MOV *direct*,A	0xF5	2	1	None

MOV copies the value of *operand2* into *operand1*. The value of *operand2* is not affected.

See Also: MOVC, MOVX, XCH, XCHD, PUSH, POP

MOV – Move into/out of Carry Bit

Syntax: MOV *bit1*,*bit2*

Instructions	OpCode	Bytes	Cycles	Flags
MOV C,*bitAddr*	0xA2	2	1	C
MOV *bitAddr*,C	0x92	2	2	None

MOV copies the value of *bit2* into *bit1*. The value of *bit2* is not affected. Either *bit1* or *bit2* must refer to the Carry bit.

MOV – Move into/out of Internal RAM

Syntax: MOV *operand1*,*operand2*

Instructions	OpCode	Bytes	Cycles	Flags
MOV @R0,#*data8*	0x76	2	1	None
MOV @R1,#*data8*	0x77	2	1	None
MOV @R0,*direct*	0xA6	2	2	None
MOV @R1,*direct*	0xA7	2	2	None
MOV R0,#*data8*	0x78	2	1	None
MOV R1,#*data8*	0x79	2	1	None
MOV R2,#*data8*	0x7A	2	1	None
MOV R3,#*data8*	0x7B	2	1	None
MOV R4,#*data8*	0x7C	2	1	None
MOV R5,#*data8*	0x7D	2	1	None
MOV R6,#*data8*	0x7E	2	1	None
MOV R7,#*data8*	0x7F	2	1	None
MOV R0,*direct*	0xA8	2	2	None
MOV R1,*direct*	0xA9	2	2	None
MOV R2,*direct*	0xAA	2	2	None
MOV R3,*direct*	0xAB	2	2	None
MOV R4,*direct*	0xAC	2	2	None
MOV R5,*direct*	0xAD	2	2	None
MOV R6,*direct*	0xAE	2	2	None
MOV R7,*direct*	0xAF	2	2	None
MOV *direct*,#*data8*	0x75	3	2	None
MOV *direct*,@R0	0x86	2	2	None
MOV *direct*,@R1	0x87	2	2	None
MOV *direct*,R0	0x88	2	2	None
MOV *direct*,R1	0x89	2	2	None
MOV *direct*,R2	0x8A	2	2	None
MOV *direct*,R3	0x8B	2	2	None
MOV *direct*,R4	0x8C	2	2	None
MOV *direct*,R5	0x8D	2	2	None
MOV *direct*,R6	0x8E	2	2	None
MOV *direct*,R7	0x8F	2	2	None
MOV *direct1*,*direct2*	0x85	3	2	None

MOV copies the value of *operand2* into *operand1*. The value of *operand2* is not affected.

NOTE: In the case of "MOV direct1,direct2 ", the operand bytes of the instruction are stored in reverse order. That is, the instruction consisting of the bytes 85h, 20h, 50h means "Move the contents of Internal RAM location 0x20 to Internal RAM location 0x50" whereas the opposite would be generally presumed.

See Also: MOVC, MOVX, XCH, XCHD, PUSH, POP

MOV DPTR – Move value into DPTR

Syntax: MOV DPTR,#*data16*

Instructions	OpCode	Bytes	Cycles	Flags
MOV DPTR,#*data16*	0x90	3	2	None

Sets the value of the Data Pointer (DPTR) to the value *data16*.

See Also**: MOVX, MOVC**

MOVC – Move Code Byte to Accumulator

Syntax: MOVC A,@A+register

Instructions	OpCode	Bytes	Cycles	Flags
MOVC A,@A+DPTR	0x93	1	2	None
MOVC A,@A+PC	0x83	1	1	None

MOVC moves a byte from code memory into the Accumulator. The code memory address from which the byte will be moved is calculated by summing the value of the Accumulator with either DPTR or the Program Counter (PC). In the case of the Program Counter, PC is first incremented by 1 before being summed with the Accumulator.

See Also: MOV, MOVX

MOVX – Move Data to/from External RAM

Syntax: MOVX operand1,operand2

Instructions	OpCode	Bytes	Cycles	Flags
MOVX @DPTR,A	0xF0	1	2	None
MOVX @R0,A	0xF2	1	2	None
MOVX @R1,A	0xF3	1	2	None
MOVX A,@DPTR	0xE0	1	2	None
MOVX A,@R0	0xE2	1	2	None
MOVX A,@R1	0xE3	1	2	None

MOVX moves a byte to or from external memory into or from the Accumulator.

If *operand1* is @DPTR, the Accumulator is moved to the 16-bit External Memory address indicated by DPTR. This instruction uses both P0 (port 0) and P2 (port 2) to output the 16-bit address and data. If *operand2* is DPTR then the byte is moved from external memory into the Accumulator.

If *operand1* is @R0 or @R1, the Accumulator is moved to the 8-bit external memory address indicated by the specified register. This instruction uses only P0 (port 0) to output the 8-bit address and data. P2 (port 2) is not affected. If *operand2* is @R0 or @R1 then the byte is moved from external memory into the Accumulator.

See Also: MOV, MOVC

MUL – Multiply Accumulator by B

Syntax: MUL AB

Instructions	OpCode	Bytes	Cycles	Flags
MUL AB	0xA4	1	4	C, OV

Multiplies the unsigned value in the Accumulator by the unsigned value in the "B" register. The least-significant byte of the result is placed in the Accumulator and the most-significant-byte is placed in the "B" register.

The **Carry Flag** (**C**) is always cleared.

The **Overflow Flag** (**OV**) is set if the result is greater than 255 (if the most-significant byte is not zero), otherwise it is cleared.

See Also: DIV

NOP – No Operation

Syntax: NOP

Instructions	OpCode	Bytes	Cycles	Flags
NOP	0x00	1	1	None

NOP, as it's name suggests, causes no operation to take place for one machine cycle. NOP is generally used only for timing purposes. Absolutely no flags or registers are affected.

ORL – Bitwise OR

Syntax: ORL operand1,operand2

Instructions	OpCode	Bytes	Cycles	Flags
ORL *direct*,A	0x42	2	1	None
ORL *direct*,*#data8*	0x43	3	2	None
ORL A,*#data8*	0x44	2	1	None
ORL A,*direct*	0x45	2	1	None
ORL A,@R0	0x46	1	1	None
ORL A,@R1	0x47	1	1	None
ORL A,R0	0x48	1	1	None
ORL A,R1	0x49	1	1	None
ORL A,R2	0x4A	1	1	None
ORL A,R3	0x4B	1	1	None
ORL A,R4	0x4C	1	1	None
ORL A,R5	0x4D	1	1	None
ORL A,R6	0x4E	1	1	None
ORL A,R7	0x4F	1	1	None
ORL C,bitAddr	0x72	2	2	C
ORL C,*/bitAddr*	0xA0	2	1	C

ORL does a bitwise "OR" operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of operand2 is not affected. A logical "OR" compares the bits of each operand and

sets the corresponding bit in the resulting byte if the bit was set in either of the original operands, otherwise the resulting bit is cleared.

See Also: ANL, XRL

POP – Pop Value from Stack

Syntax: POP *register*

Instructions	OpCode	Bytes	Cycles	Flags
POP *direct*	0xD0	2	2	None

POP "pops" the last value placed on the stack into the *direct* address specified. In other words, POP will load *direct* with the value of the Internal RAM address pointed to by the current Stack Pointer. The stack pointer is then decremented by 1.

NOTE #1: The address of direct must be an Internal RAM or SFR address. You cannot POP directly into "R" registers, such as R0, R1, etc. To pop a value off the stack into R0, for example, you must pop the value into the accumulator and then move the value of the accumulator into R0.

NOTE #2: When popping a value off the stack into the Accumulator, you must code the instruction as **POP ACC**, not **POP A**. The latter is invalid and will result in an error at assemble-time.

See Also: PUSH

PUSH – Push Value onto Stack

Syntax: PUSH *register*

Instructions	OpCode	Bytes	Cycles	Flags
PUSH *direct*	0xC0	2	2	None

PUSH "pushes" the value of the specified *direct* address onto the stack. PUSH first increments the value of the Stack Pointer by 1, then takes the value stored in *direct* and stores it in internal RAM at the location pointed to by the incremented Stack Pointer.

NOTE #1: The address of direct must be an Internal RAM or SFR address. You cannot PUSH directly from "R" registers, such as R0, R1, etc. To push a value onto the stack from R0, for example, you must move R0 into the accumulator, then PUSH the value of the accumulator onto the stack.

NOTE #2: When pushing a value from the accumulator onto the stack into the, you must code the instruction as **PUSH ACC**, not **PUSH A**. The latter is invalid and will result in an error at assemble-time.

See Also: POP

RET – Return from Subroutine

Syntax: RET

Instructions	OpCode	Bytes	Cycles	Flags
RET	0x22	1	2	None

RET is used to return from a subroutine previously called by LCALL or ACALL. Program execution continues at the address that is calculated by popping the top-most 2 bytes off the stack. The most-significant-byte is popped off the stack first, followed by the least-significant-byte.

See Also: LCALL, ACALL, RETI

RETI – Return from Interrupt

Syntax: RETI

Instructions	OpCode	Bytes	Cycles	Flags
RETI	0x32	1	2	None

RETI is used to return from an interrupt service routine. RETI first enables interrupts of equal and lower priorities to the interrupt that is terminating. Program execution continues at the address that is calculated by popping the top-most 2 bytes off the stack. The most-significant-byte is popped off the stack first, followed by the least-significant-byte.

RETI functions identically to RET if it is executed outside of an interrupt service routine.

See Also: RET

RL – Rotate Accumulator Left

Syntax: RL A

Instructions	OpCode	Bytes	Cycles	Flags
RL A	0x23	1	1	C

Shifts the bits of the accumulator to the left. The left-most bit (bit 7) of the Accumulator is loaded into bit 0.

See Also: RLC, RR, RRC

RLC – Rotate Accumulator Left Through Carry

Syntax: RLC A

Instructions	OpCode	Bytes	Cycles	Flags
RLC A	0x33	1	1	C

Shifts the bits of the accumulator to the left. The left-most bit (bit 7) of the accumulator is loaded into the Carry Flag, and the original Carry Flag is loaded into bit 0 of the Accumulator.

See Also: RL, RR, RRC

RR – Rotate Accumulator Right

Syntax: RR A

Instructions	OpCode	Bytes	Cycles	Flags
RR A	0x03	1	1	None

Shifts the bits of the accumulator to the right. The right-most bit (bit 0) of the accumulator is loaded into bit 7.

See Also: RL, RLC, RRC

RRC – Rotate Accumulator Right Through Carry

Syntax: RRC A

Instructions	OpCode	Bytes	Cycles	Flags
RRC A	0x13	1	1	C

Shifts the bits of the accumulator to the right. The right-most bit (bit 0) of the accumulator is loaded into the Carry Flag, and the original Carry Flag is loaded into bit 7.

See Also: RL, RLC, RR

SETB – Set Bit

Syntax: SETB *bitAddr*

Instructions	OpCode	Bytes	Cycles	Flags
SETB C	0xD3	1	1	C
SETB *bitAddr*	0xD2	2	1	None

Sets the specified bit.

If the instruction requires the Carry bit to be set, the assembler will automatically use the 0xD3 opcode. If any other bit is set, the assembler will automatically use the 0xD2 opcode.

See Also: CLR

SJMP – Short Jump

Syntax: SJMP *relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
SJMP *relAddr*	0x80	2	2	None

SJMP jumps unconditionally to the address specified *relAddr*. *RelAddr* must be within -128 or +127 bytes of the instruction that follows the SJMP instruction.

See Also: LJMP, AJMP

SUBB – Subtract from Accumulator with Borrow

Syntax: SUBB A,*operand*

Instructions	OpCode	Bytes	Cycles	Flags
SUBB A,#*data8*	0x94	2	1	C, AC, OV
SUBB A,*direct*	0x95	2	1	C, AC, OV
SUBB A,@R0	0x96	1	1	C, AC, OV
SUBB A,@R1	0x97	1	1	C, AC, OV
SUBB A,R0	0x98	1	1	C, AC, OV
SUBB A,R1	0x99	1	1	C, AC, OV
SUBB A,R2	0x9A	1	1	C, AC, OV
SUBB A,R3	0x9B	1	1	C, AC, OV
SUBB A,R4	0x9C	1	1	C, AC, OV
SUBB A,R5	0x9D	1	1	C, AC, OV
SUBB A,R6	0x9E	1	1	C, AC, OV
SUBB A,R7	0x9F	1	1	C, AC, OV

SUBB subtracts the value of *operand* from the value of the accumulator, leaving the resulting value in the accumulator. The value *operand* is not affected.

The **Carry Bit** (**C**) is set if a borrow was required for bit 7, otherwise it is cleared. In other words, if the unsigned value being subtracted is greater than the accumulator the carry flag is set.

The **Auxillary Carry** (**AC**) bit is set if a borrow was required for bit 3, otherwise it is cleared. In other words, the bit is set if the low nibble of the value being subtracted was greater than the low nibble of the accumulator.

The **Overflow** (**OV**) bit is set if a borrow was required for bit 6 or for bit 7, but not both. In other words, the subtraction of two signed bytes resulted in a value outside the range of a signed byte (-128 to 127). Otherwise it is cleared.

See Also: ADD, ADDC, DEC

SWAP – Subtract Accumulator Nibbles

Syntax: SWAP A

Instructions	OpCode	Bytes	Cycles	Flags
SWAP A	0xC4	1	1	None

SWAP swaps bits 0-3 of the Accumulator with bits 4-7 of the Accumulator. This instruction is identical to executing "RR A" or "RL A" four times.

See Also: RL, RLC, RR, RRC

XCH – Exchange Bytes

Syntax: XCH A,*register*

Instructions	OpCode	Bytes	Cycles	Flags
XCH A,@R0	0xC6	1	1	None
XCH A,@R1	0xC7	1	1	None
XCH A,R0	0xC8	1	1	None
XCH A,R1	0xC9	1	1	None
XCH A,R2	0xCA	1	1	None
XCH A,R3	0xCB	1	1	None
XCH A,R4	0xCC	1	1	None
XCH A,R5	0xCD	1	1	None
XCH A,R6	0xCE	1	1	None
XCH A,R7	0xCF	1	1	None
XCH A,*direct*	0xC5	2	1	None

Exchanges the value of the accumulator with the value contained in *register*.

See Also: MOV

XCHD – Exchange Digit

Syntax: XCHD A,*register*

Instructions	OpCode	Bytes	Cycles	Flags
XCHD A,@R0	0xD6	1	1	None
XCHD A,@R1	0xD7	1	1	None

Exchanges bits 0-3 of the accumulator with bits 0-3 of the Internal RAM address pointed to indirectly by R0 or R1. Bits 4-7 of each register are unaffected.

See Also: DA

XRL – Bitwise Exclusive OR

Syntax: XRL operand1,operand2

Instructions	OpCode	Bytes	Cycles	Flags
XRL *direct*,A	0x62	2	1	None
XRL *direct*,#*data8*	0x63	3	2	None
XRL A,#*data8*	0x64	2	1	None
XRL A,*direct*	0x65	2	1	None
XRL A,@R0	0x66	1	1	None
XRL A,@R1	0x67	1	1	None
XRL A,R0	0x68	1	1	None
XRL A,R1	0x69	1	1	None
XRL A,R2	0x6A	1	1	None
XRL A,R3	0x6B	1	1	None
XRL A,R4	0x6C	1	1	None
XRL A,R5	0x6D	1	1	None
XRL A,R6	0x6E	1	1	None
XRL A,R7	0x6F	1	1	None

XRL does a bitwise "EXCLUSIVE OR" operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of *operand2* is not affected. A logical "EXCLUSIVE OR" compares the bits of each operand and sets the corresponding bit in the resulting byte if the bit was set in either (but not both) of the original operands, otherwise the bit is cleared.

See Also: ANL, ORL

UNDEFINED – Undefined Instruction

Syntax: ???

Instructions	OpCode	Bytes	Cycles	Flags
???	0xA5	1	1	C

The "undefined" instruction is, as the name suggests, not a documented instruction. The 8052 supports 255 instructions and OpCode 0xA5 is the single OpCode that is not used by any documented function. Since it is not documented nor defined it is not recommended that it be executed.

However, based on my research, executing this undefined instruction takes 1 machine cycle and appears to have no effect on the system except that the Carry Bit always seems to be set.

NOTE: We received input from an 8052.com user that the undefined instruction really has a format of **Undefined bit1,bit2** and effectively copies the value of *bit2* to *bit1*. In this case, it would be a three-byte instruction. We haven't had an opportunity to verify or disprove this report, so we present it to the world as "additional information."

See Also: NOP

Appendix C: SFR Quick Reference

Name	Addr	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
B*	F0	*No bit-level significance*							
ACC*	E0	*No bit-level significance*							
PSW*	D0	CY	AC	F0	RS1	RS0	OV	-	P
IP*	B8	-	-	PT2	PS	PT1	PX1	PT0	PX0
P3*	B0	RD	WR	T1	T0	INT1	INT0	TXD	RXD
IE*	A8	EA	-	ET2	ES	ET1	EX1	ET0	EX0
P2*	A0	A15	A14	A13	A12	A11	A10	A9	A8
SBUF	99	*No bit-level significance*							
SCON*	98	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
P1*	90	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
TH1	8D	*No bit-level significance*							
TH0	8C	*No bit-level significance*							
TL1	8B	*No bit-level significance*							
TL0	8A	*No bit-level significance*							
TMOD	89	GATE_1	C/T_1	M1_1	M0_1	GATE_0	C/T_0	M1_0	M0_0
TCON*	88	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
PCON	87	SMOD	-	-	-	GF1	GF0	PD	IDL
DPH	83	*No bit-level significance*							
DPL	82	*No bit-level significance*							
SP	81	*No bit-level significance*							
P0*	80	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0

(*) = SFR which is bit-addressable. SFRs without asterisk are *not* bit-addressable.

NOTE: The bit names assigned to P0 and P2 refer to their function at the hardware level. P0.0 is named "AD0" because its function is Address/Data Line 0. However, the names assigned to the bits of P0 and P2 will not be recognized as valid by a standard 8052 assembler unless you specifically define a symbol in your program as referencing those bits.

Appendix D: SFR Detailed Reference (Alphabetical)

INTERRUPT PRIORITY (IE)

SFR Name: IE
SFR Address: A8h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	EA	-	ET2	ES	ET1	EX1	ET0	EX0
BIT ADDR	AFh	AEh	ADh	ACh	ABh	AAh	A9h	A8h

EA – Enable/Disable All Interrupts. 1=Interrupts enabled

ET2 – Enable Timer 2 Interupt. 1=Interrupt enabled

ES – Enable Serial Interupt. 1=Interrupt enabled

ET1 – Enable Timer 1 Interupt. 1=Interrupt enabled

EX1 – Enable External 1 Interupt. 1=Interrupt enabled

ET0 – Enable Timer 0 Interupt. 1=Interrupt enabled

EX0 – Enable External 0 Interupt. 1=Interrupt enabled

INTERRUPT PRIORITY (IP)

SFR Name: IP
SFR Address: B8h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	-	-	PT2	PS	PT1	PX1	PT0	PX0
BIT ADDR	BFh	BEh	BDh	BCh	BBh	BAh	B9h	B8h

PT2 – Priority Timer 2 Interupt. 1=High priority interrupt, 0=Low priority interrupt

PS – Priority Serial Interupt. 1=High priority interrupt, 0=Low priority interrupt

PT1 – Priority Timer 1 Interupt. 1=High priority interrupt, 0=Low priority interrupt

PX1 – Priority External 1 Interupt. 1=High priority interrupt, 0=Low priority interrupt

PT0 – Priority Timer 0 Interupt. 1=High priority interrupt, 0=Low priority interrupt

PX0 – Priority External 0 Interupt. 1=High priority interrupt, 0=Low priority interrupt

PORT 0 (P0)

SFR Name: P0
SFR Address: 80h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
BIT ADDR	87h	86h	85h	84h	83h	82h	81h	80h

NOTE: These bit names indicate that I/O line's function on the P0 bus when used with external memory (code/RAM). A standard 8052 assembler will not recognize these bits by the given names, rather they'll only be recognized as P0.7, P0.6, etc.

NOTE: Port 0 is only available for general input/output if the project does not use external code memory or external RAM. When such external memory is used, port 0 is used automatically by the microcontroller to address the memory and read/write data from/to said memory.

PORT 1 (P1)

SFR Name: P1
SFR Address: 90h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	T2EX	T2
BIT ADDR	97h	96h	95h	94h	93h	92h	91h	90h

T2EX – Timer 2 Capture/Reload. Optional external capturing or reloading of timer 2.

T2 – Timer 2 External Input. Optionally used to control timer/counter 2 via external source.

PORT 2 (P2)

SFR Name: P2
SFR Address: A0h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	A15	A14	A13	A12	A11	A10	A9	A8
BIT ADDR	A7h	A6h	A5h	A4h	A3h	A2h	A1h	A0h

NOTE: These bit names indicate that I/O line's function on the P2 bus when used with external memory (code/RAM). A standard 8052 assembler will not recognize these bits by the given names, rather they'll only be recognized as P2.7, P2.6, etc.

NOTE: Port 2 is only available for general input/output if the project does not use external code memory or external RAM. When such external memory is used, port 2 is used automatically by the microcontroller to address the memory and read/write data from/to said memory.

PORT 3 (P3)

SFR Name: P3
SFR Address: B0h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	RD	WR	T1	T0	INT1	INT0	TXD	RXD
BIT ADDR	B7h	B6h	B5h	B4h	B3h	B2h	B1h	B0h

RD – Read Strobe. 0=External memory read strobe.

WR – Write Strobe. 0=External memory write strobe.

T1 – Timer/Counter 1 External Input. Optionally used to control timer/counter 1 via external source.

T0 – Timer/Counter 0 External Input. Optionally used to control timer/counter 0 via external source.

INT1 – External Interrupt 1. Used to trigger external interrupt 1.

INT0 – External Interrupt 0. Used to trigger external interrupt 0.

TXD – Serial Transmit Data. 8052's serial transmit line (from 8052 to external device).

RXD – Serial Transmit Data. 8052's serial receive line (to 8052 from external device).

NOTE: These bit names indicate that I/O line's function on the P3 bus. A standard 8052 assembler will not recognize these bits by the given names, rather they'll only be recognized as P3.7, P3.6, etc.

POWER STATUS WORD (PSW)

SFR Name: PSW
SFR Address: D0h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	CY	AC	F0	RS1	RS0	OV	-	P
BIT ADDR	D7h	D6h	D5h	D4h	D3h	D2h	D1h	D0h

CY – Carry Flag. Set or cleared by instructions ADD, ADDC, SUBB, MUL, and DIV.

AC – Auxiliary Carry. Set or cleared by instructions ADD, ADDC.

F0 – Flag 0. General flag available to developer for user-defined purposes.

RS1/RS0 – Register Select Bits. These two bits, taken together, select the register bank which will be used when using "R" registers R0 through R7, according to the following table:

RS1	RS0	Register Bank	Register Bank Addresses
0	0	0	00h – 07h
0	1	1	08h – 0Fh
1	0	2	10h – 17h
1	1	3	18h – 1Fh

OV – Overflow Flag. Set or cleared by instructions ADD, ADDC, SUBB, and DIV.

P – Parity Flag. Set or cleared automatically by core to establish even parity with the accumulator, such that the number of bits set in the accumulator plus the value of the parity bit will always equal an even number.

SERIAL CONTROL (SCON)

SFR Name: SCON
SFR Address: 98h
Bit-Addressable: Yes
Bit-Definitions:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NAME	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
BIT ADDR	9Fh	9Eh	9Dh	9Ch	9Bh	9Ah	99h	98h

SM0/SM1– Serial Mode. These two bits, taken together, select the serial mode in which the serial port will operate.

SM0	SM1	SERIAL MODE	DESCRIPTION	BAUD
0	0	0	Shift Register	Oscillator / 12
0	1	1	8-bit UART	Variable (T1 or T2)
1	0	2	9-bit UART	Oscillator / 64 or /32
1	1	3	9-bit UART	Variable (T1 or T2)

SM2 – Serial Mode 2 (Multiprocessor Communication). When this bit is set, multiprocessor communication is enabled in modes 2 and 3 causing the RI bit to only be set when the 9th bit of a byte received is set. In mode 1, RI will only be set if a valid stop bit is received. SM2 should be cleared in mode 0.

REN – Received Enable. This bit must be set to enable data reception via the serial port. No data will be received by the serial port if this bit is clear.

TB8– Transmit Bit 8. When in modes 2 and 3, this bit will be the 9th bit that is sent when a byte is written to SBUF.

RB8– Receive Bit 8. When in modes 2 and 3, this is the 9th bit that was received. In mode 1, and if SM2 is set, RB8 holds the value of the stop bit that was received. RB8 is not used in mode 0.

TI – Transmit Interrupt. Set by hardware when the byte previously written to SBUF has been completely clocked out the serial port.

RI – Receive Interrupt. Set by hardware when a byte has been received by the serial port and is availble to be read in SBUF.