2

8051 PROCESSOR ARCHITECTURE

CONTENTS AT A GLANCE

The 8051 processor architecture is Harvard-based with external memory read/write capabilities built in as part of the architecture. The 8051's architecture is really a "typical" MCU architecture. If you compare the architecture to the AVR or the PIC, you shouldn't have any problems with understanding how the 8051 works relative to these parts. However, if you are expecting the 8051 to work like the 8080 or 8086 (i.e., Intel microprocessors), you are in for quite a shock.

The CPU

The basic 8051 architecture, from a high level, isn't all that different from the other Harvard architectures presented in this book. With the architecture diagram shown in Fig. 2-1, the basic 8051 should look like the Harvard architecture I presented earlier in the book.

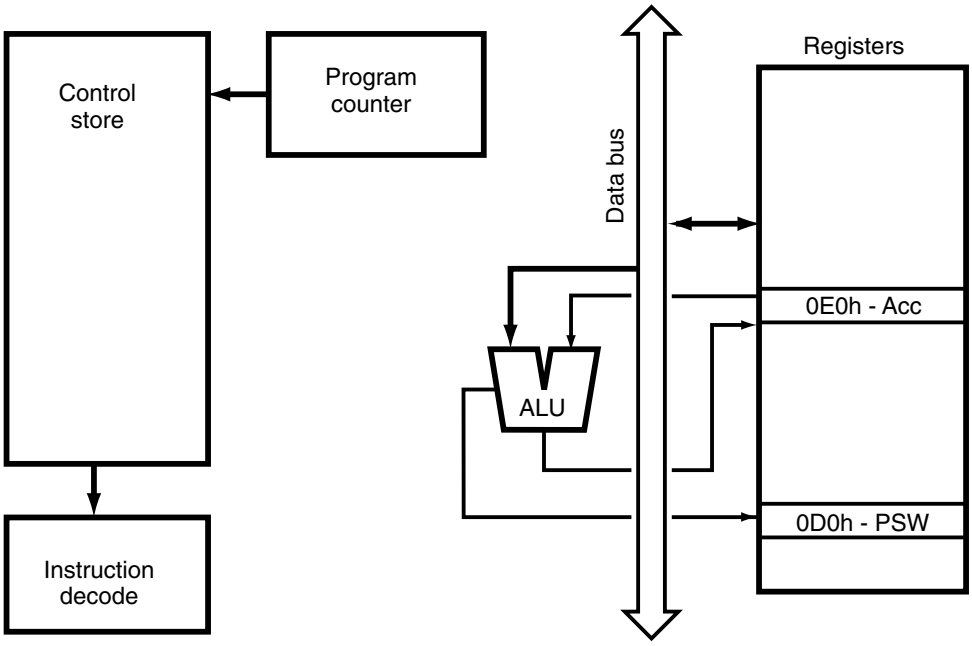

FIGURE 2-1 Basic 8051 architecture.

For the most part, the 8051 isn't a difficult device to program. However, there are a few quirks that you need to understand before you will feel comfortable with the design. The two areas that I will expand upon are the memory organization in general (including the control store and register boxes in Fig. 2-1) and how registers are implemented specifically.

To explain how the registers work, I'm going to apply a magnifying glass to the register box from Fig. 2-1. (See Fig. 2-2.) I have broken up the first 256 addresses by how the registers are accessed by *direct addressing* (which is when the register address is specifically given in the instruction) and *indirect addressing* (when the register address is located within an index register).

You might note that I have marked the direct addresses 020h to 02Fh differently in Fig. 2-2. These 16 bytes can be accessed as 128 bits as well as bytes. Some of the bits in the special-purpose registers (in address 080h and above) can be accessed directly, although for others they could be loaded into the bit addressable area (addresses 020h to 02Fh), modified, then stored back into the special-purpose (I/O) register area if individual bits were to be modified. The 128 bits can also be accessed as 16 regular RAM bytes.

If you directly address 080h to 0FFh, you will be accessing the special-purpose (I/O) register area of the 8051. This register area is reserved for processor and I/O peripheral registers. (See Fig. 2-3.)

The shaded areas are addresses that might or might not have hardware registers for controlling peripheral functions. The areas that aren't shaded are the standard registers that are available in all 8051s. I have refrained from extending the standard register definitions to anything more than an absolute minimum because, with all the different

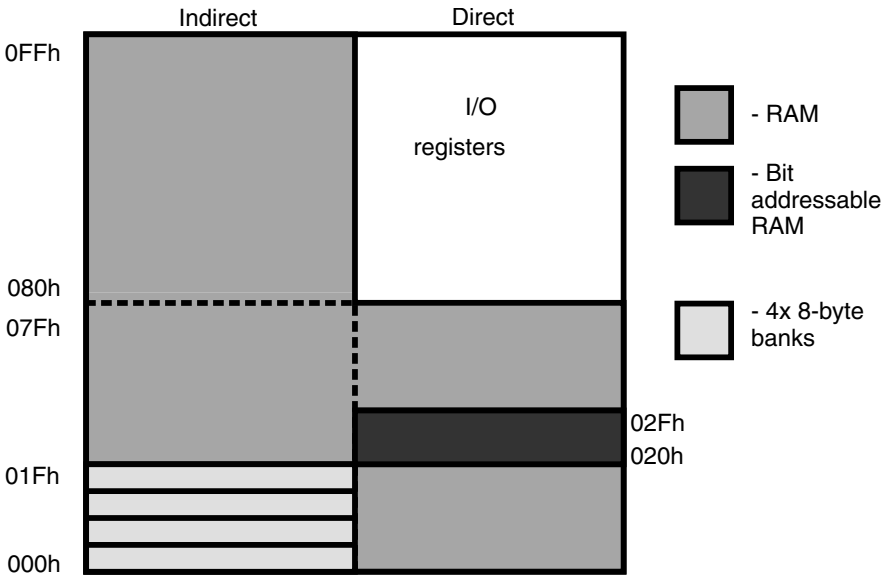

FIGURE 2-2 8051 address 0-0FFh registers.

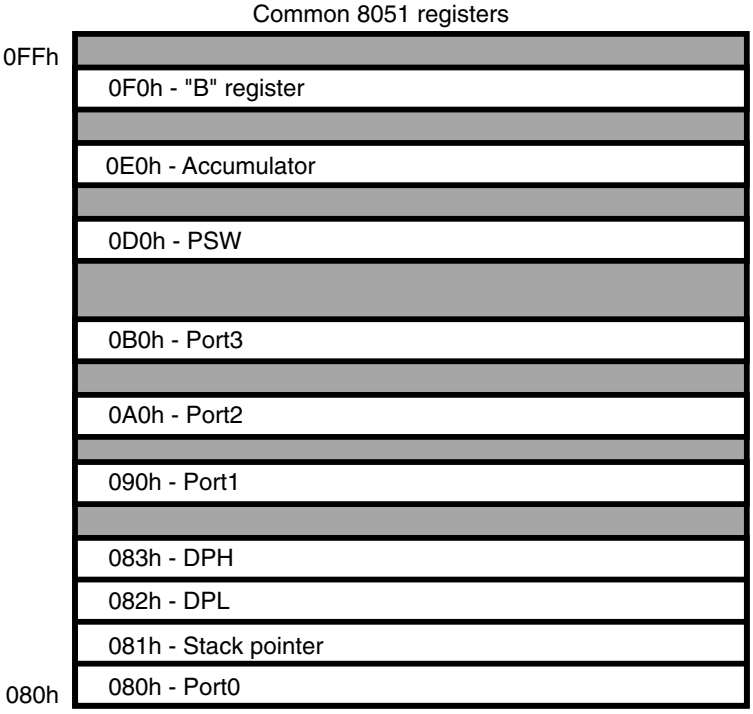

FIGURE 2-3 Standard 8051 register addresses.

8051 manufacturers, there are a number of standard peripherals and their associated registers that are not available in every device.

The special function registers I have identified are: the stack pointer, index pointer (DPL and DPH), I/O port addresses, the status register (program status word or PSW), and accumulators. I will present more information on these registers later in the book.

All direct addressing can only access the first 256 addresses in the memory space (which is marked as "Registers" in Fig. 2-1). To access RAM bytes at addresses above this 256 address space, you'll have to use an index register (such as DPTR, which is made up of DPL and DPH).

Indirect (or *indexed*) *addressing* uses either the stack pointer (the SP register identified earlier) or an index register (DPTR, for example). The special function registers (at address 080h to 0FFh) cannot be accessed by the indirect addressing. The 8052 enhancements have put in 128 bytes of RAM that can only be accessed here by the stack pointer or an index pointer.

With RAM put into the indirect space at the same addresses as the "special function registers" something funny starts to happen: You can access 384 different RAM bytes and I/O registers in the first 256 addresses of the 8051. This is 256 addresses of RAM (which can only be fully accessed by using indirect addressing instructions) and 128 addresses of special function registers (which can only be accessed using direct addressing instructions). When I first read this in the Dallas Semiconductor's HSM documentation, I really ended up shaking my head.

With this information on the register area, I can now expand the basic architecture block diagram to that shown in Fig. 2-4.

In describing the special purpose registers, I think I left a few open questions. The first is: What are all the special purpose registers that I identified? I will go through the processor-specific registers here, but leave the I/O specific registers for later in this section.

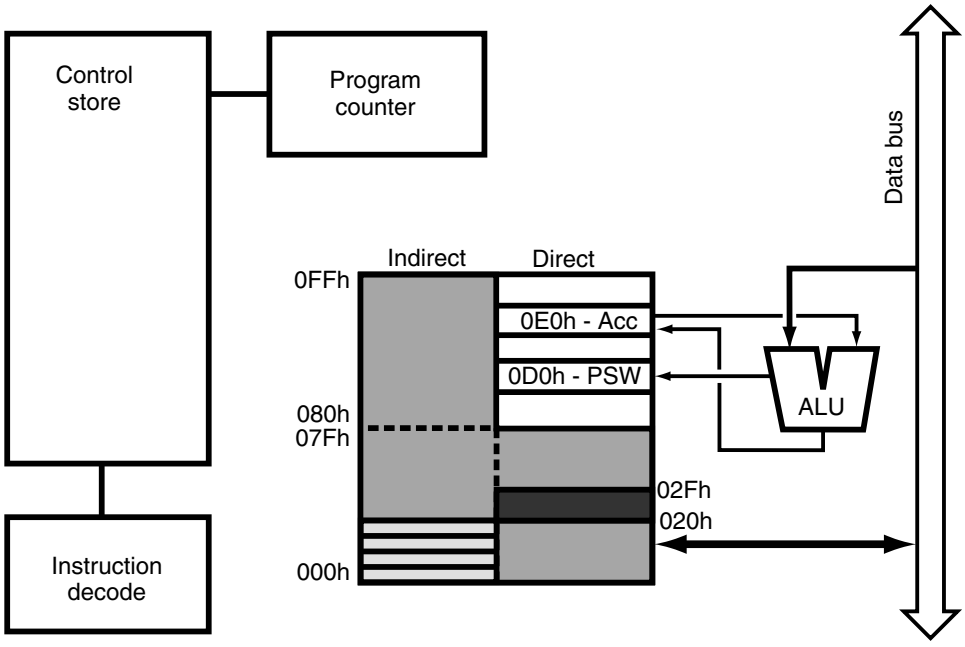

FIGURE 2-4 8051 architecture with register addressing.

TABLE 2-1 DEFINITION OF THE 8051 PSW

BIT	PSW BIT FUNCTION
7	C (carry flag)—Set when addition > 0FFh or when subtraction < 0
6	AC (auxillary carry flag)—Set when the low nybble affects the high nybble
5	F0 (user flag 0)
4	RS1 (register bank select 1)
3	RS0 (register bank select 0)
2	OV (overflow flag)—Set after addition or subtraction, cleared by all others
1	F1 (user flag 1)
0	P (parity flag)—Set when the accumulator has an odd number of bits

The accumulator registers (A and B at addresses 0E0h and 0F0h, respectively) are used to store temporary values and the results of arithmetic operations. If you flip through the arithmetic and bitwise instructions quickly, you'll see that B is just about never used. It is only used for multiplication and division. The accumulators can be accessed either as part of an instruction or as addresses 0E0h or 0F0h for ACC (which is the special function register identification for the accumulator) and B, respectively.

The program status word (most commonly known as PSW at address 0D0h) is the status register for the 8051. It is defined in Table 2-1.

The CY and AC bits are the full carry and half carry flags. CY is set when an addition or subtraction causes a carry to or a borrow from the next highest byte of the number (which allows easy manipulation of 16 bit or larger values). The AC, or half carry, flag is set when an addition or subtraction makes the lower nybble change the value (by one) of the upper nybble. These two status flags are in all the microcontrollers presented in this book.

Looking at the PSW definition, there is one flag that you would expect to see but don't. That is the zero flag (which is set when the result of an arithmetic, bitwise, or shift/rotate operation is equal to zero). Checking for zero is handled differently in the 8051. The `jz` instruction tests the contents of the accumulator (A) and carries out the jump if the contents of the register is equal to zero. I will expand on this operation in chapter 3.

The lack of a zero flag is something I've pondered about, wondering whether it's good or not. In many ways, it is a positive (most notably in helping to keep interrupt handlers short and not unnecessarily pushing and popping the PSW on and off the stack). I guess it seems foreign to me not to have one and takes some getting used to (and learning how to think through developing applications for it).

The RS0 and RS1 bits are used to select which of the four 8-byte banks is currently being used. These 8-byte banks are used for providing single-byte arithmetic instructions. By providing a very small area to operate out of, smaller and faster instructions can be used in your application. One aspect of the 8-byte banks that you should be aware of is that the two least-significant bytes of the bank (identified as R0 and R1) can be used for index addressing in the 256-address RAM area.

Another item I have not addressed so far is the operation of the stack pointer and how it can only access the first 256 addresses of the RAM register area. Upon power up, it is set to 007h, but when using an 8052, I like to change it to 080h and just forget about worrying how much space is required for the stack.

This might seem a bit strange, and there are a few things I really haven't explained here. The stack's `push` and `pop` instructions *increment* the stack pointer (rather than decrement the stack pointer as is done in many other devices). This means that, to set up the stack pointer to give it maximum space in a stack area, you have to give it the lowest address of the stack area, rather than the highest addresses, as you would with most other devices.

I set the stack pointer address to 080h because the RAM is only accessible by the index registers (which the stack pointer is one). Giving the stack pointer the bottom of the accessible area, I can put arrays at the top of the first 256 addresses and not really have to worry about whether or not the arrays or the stack will write over each other's data area.

The DPTR is a 16-bit index register that can access up to 64K different addresses in external memory. It is primarily designed for transferring data to/from the external memory area, and I'll discuss it in more detail in the next section.

With the special registers, the current block diagram looks like the one shown in Fig. 2-5. In this diagram, you can see how the stack pointer (SP), DPTR, A, and PSW registers are all accessible from the direct register area and how they interact with the 8051's register data and address busses.

With this, the block diagram of the 8051 is just about complete. The only change that I would make is adding a multiplexor selection between reading an immediate value in the instruction or using a value in the Register space. This is shown in Fig. 2-6.

With Fig. 2-6, I consider *my* block diagram of the operation of the 8051 to be complete. If you compare it to the block diagram contained in the Dallas Semiconductor's 87C520 data sheet (Fig. 2-7), you'll see just about no relation between the two diagrams. These differences

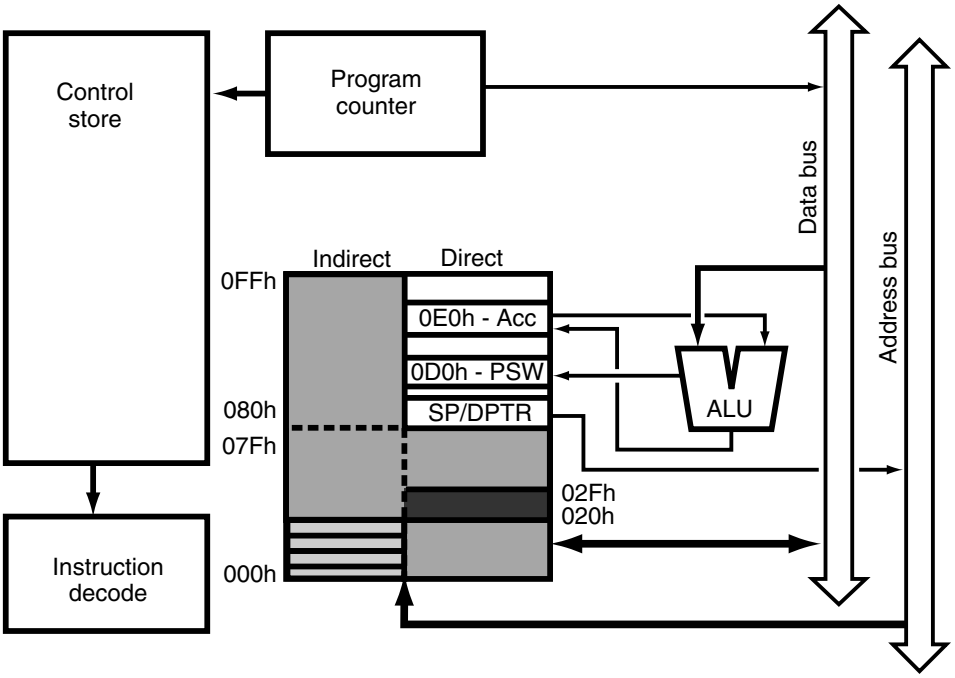

FIGURE 2-5 8051 architecture with register addressing.

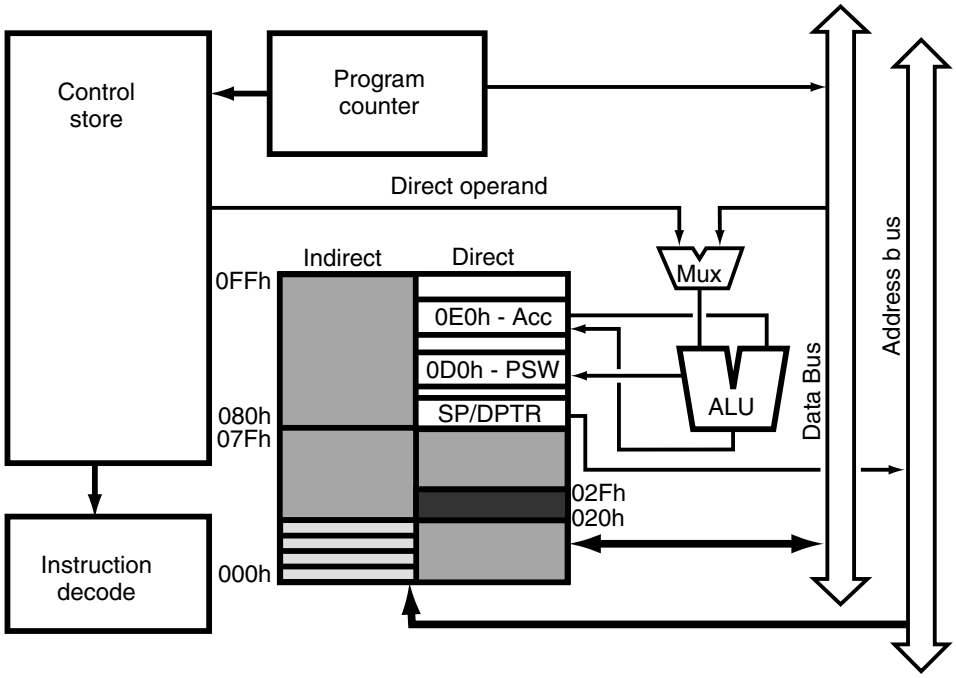

FIGURE 2-6 8051 architecture with immediate addressing.

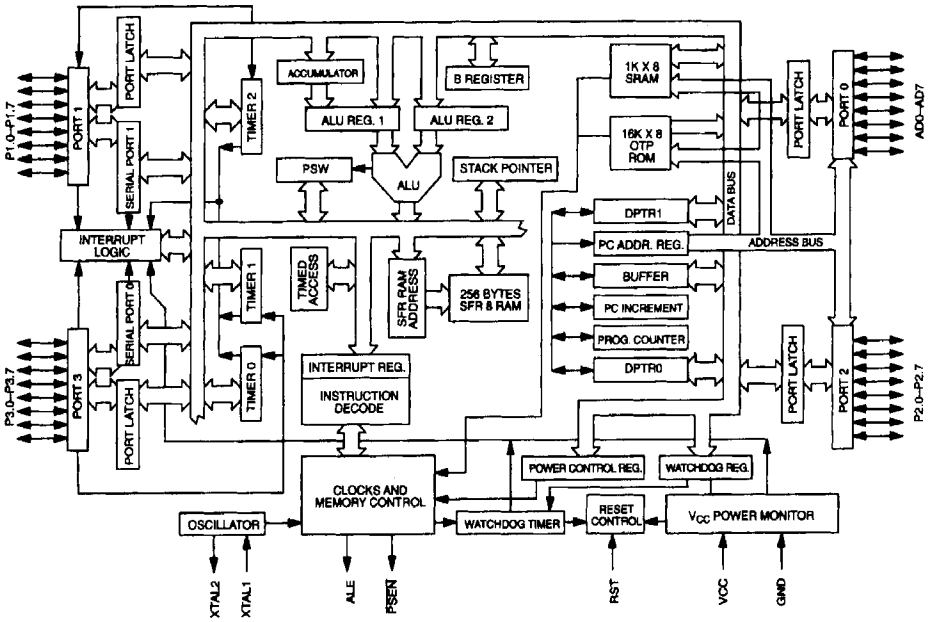

FIGURE 2-7 DS87C520 block diagram.

really come from how I try to understand how the devices work. I tend to focus on instruction execution and try to understand what has to happen inside the processor for the instruction to execute properly. My block diagrams tend to simplify how the peripheral hardware interfaces to the processor (although I will expand upon the interfaces later in each section).

Is this the right way to learn how the different microcontrollers and their processors work? I don't know, but I do know that it is a very effective way for me to learn how they work and visualize how data moves inside the devices.

8051 Addressing Modes

The 8051's addressing modes are designed from the perspective that data-access priority should reflect how the data is to be accessed. In most applications, very few variables are accessed a lot, some quite a bit, and most quite infrequently. In designing the 8051, Intel used this philosphy to specify the single-byte/12-clock-cycle instruction cycle for determining how data would be accessed. Accessing the current register banks only takes 1 byte and 12 cycles. This is followed by the 2-byte/24-clock instructions into the first 256 register addresses and the 3-byte, multiple instruction cycle access to the memory above the first 256 addresses.

This can be shown graphically in Fig. 2-8. This diagram is a good one to remember when developing complex applications in assembler. As you are writing the code, you should be asking yourself: What is the distribution of the instructions accessing the variables and is it optimized to execute in the fewest number of cycles and require the fewest number of control store bytes?

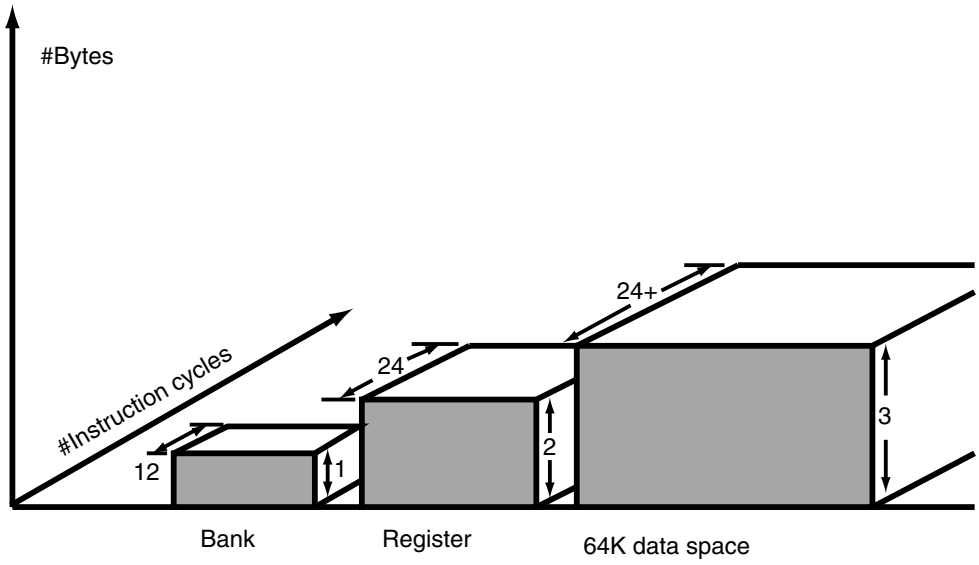

FIGURE 2-8 8051 addressing mode comparison.

The first addressing mode really isn't an addressing mode at all. *Immediate addressing* passes the value to be executed as part of the instruction. The immediate value is specified by placing a "#" character in front of the immediate value:

```
add  A, #77                    ; Add "77" to the accumulator
```

This instruction will add decimal 77 to the contents of the accumulator and store the result back into the accumulator.

Bank addressing allows you to access a byte in the current register bank. This is the most efficient (both in terms of clock cycles and control store) method of accessing data. Most register instructions execute in one register cycle and only require one byte to execute the instruction. The 8 bytes are known as R0 though R7.

Direct memory addressing differs from register addressing in that any byte within the first 256 addresses can be accessed by specifying an 8-bit address. When using this mode, there are a few things to watch out for. The first RAM addresses (080h to 0FFh, if the device you're using has RAM at these locations) cannot be accessed by direct addressing (instead you'll have to use an indirect mode). If you specify an instruction like:

```
mov  A, 088h
```

You'll load the accumulator with the contents of the TCON register (at 088h) rather than the contents of a RAM byte.

The second thing to note is something that always irritates me when I'm developing an 8051 application. I always forget to put in the "#" character for immediate addressing, and I inadvertently wind up with an instruction that uses direct addressing. This seems to be my number-one syntax problem when developing 8051 code, and it means I have to be especially careful when I'm simulating an application to make sure the correct addressing mode is being used.

Register indirect addressing uses R0 or R1 as an 8-bit index register to access a byte in the first 256 addresses of the data space:

```
orl  A, @R0
```

The register indirect is identified by the "@" character before either R0 or R1. Using any other bank registers will result in an error.

The DPTR register can also be used as a 16-bit index register. This addressing mode can also be enhanced with an offset for accessing data structures in data space memory. This mode is known as *register indirect with displacement*.

When you want to read a table out of control store (i.e., read a string of ASCII characters), the `movc A, @A+PC` instruction can access an offset from the current address. This offset has to be arithmetically generated before the instruction can be used.

The remaining addressing modes are used for changing the current program counter. You will see similar modes in the other microcontrollers. These modes are expanded upon in chapter 3. Actually, the different data addressing modes are also explained in more detail in chapter 3 as well.

External Addressing

At the start of this section, I introduced the 8051 as a device that could access up to 64K of program memory and 64K of variable SRAM. Looking through what I've shown earlier in this chapter and looking at the data sheets, you're probably wondering how this is accomplished. The 8051 is designed with a built-in external memory interface that uses the P0 and P2 I/O port bits for accesses. (See Fig. 2-9.)

Anytime an address (either control store or data space) is greater than what's in the 8051, the external I/O is activated and the 8051 tries to access external memory. If you're familar with other microcontrollers, you might wonder how this is possible with a "typical" microcontroller's I/O pin design. You might want to skip ahead and look in chapter 4 to see what I'm talking about and see how different bit states are output.

When accessing external memory, first the least significant 8 bits of the address are output, followed by either reading or writing data.

The external control store byte read waveform looks like Fig. 2-10, which is actually how the Intel 8085/8088/8086 microprocessors access external memory. For RAM, the _RD (read) and _WR (write) pins are used instead of _PSEN (which is used to access external control store). In the Dallas Semiconductor HSM 8051s, the length of time used for reading/writing data can be extended or the cycles stretched. This is analogous in the "PC universe" as adding wait states. Stretching the accesses might be required when going from a "true" 8051 to a HSM part because of the latter's faster instruction cycles or to allow "slower" devices to be accessed without violating their timing specifications.

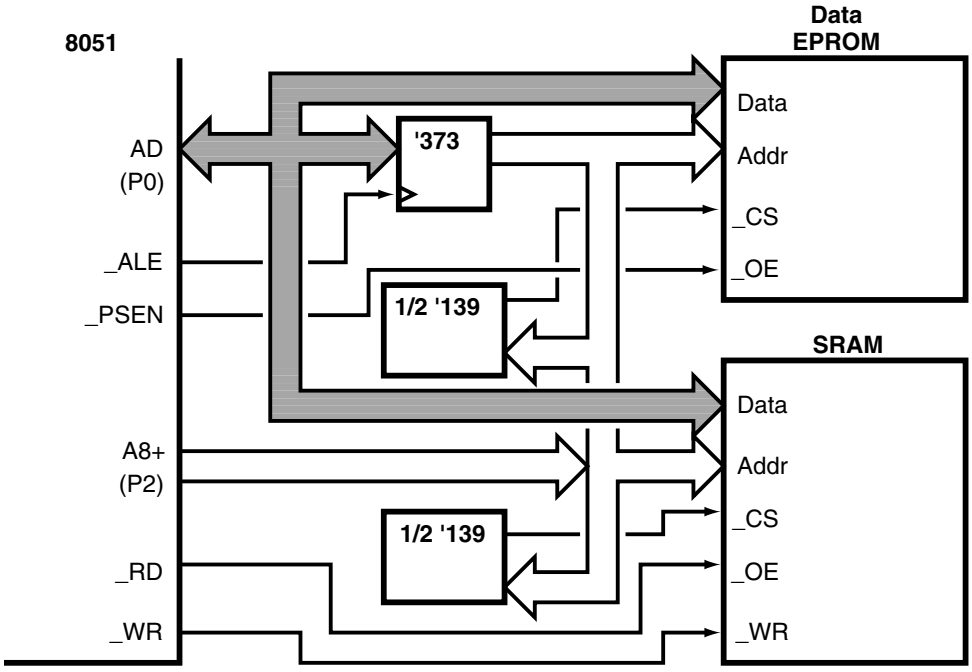

FIGURE 2-9 8051 external memory access.

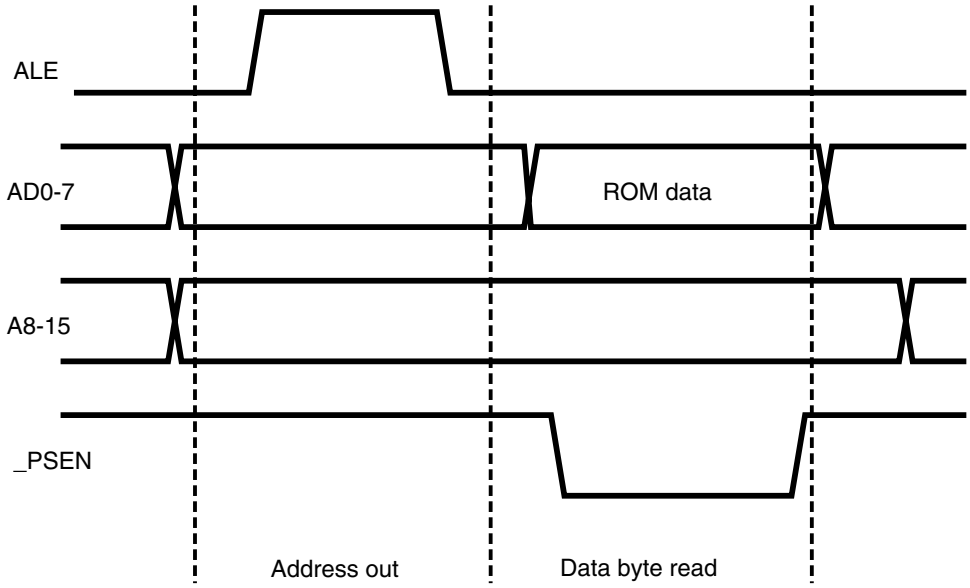

FIGURE 2-10 8051 external ROM read cycle.

Now, if you want to provide external memory (or devices) on the 8051's external busses but don't want to use the entire 64K address space, you could put these devices at high memory, which means that the I/O pins are always high and can be used for functions other than providing memory addresses.

For example, an application could be designed where 2K of SRAM was required along with two pins also required for bit I/O purposes, the 8051 and SRAM could be wired as shown in Fig. 2-11.

In this scheme, the P2.5 and P2.6 pins not used by the external RAM and are available for bit I/O because they will not change state during SRAM access. If the bit in the address is set, then its external I/O state will not change. This means that the SRAM pins should always be set high (or left at their initial value of all high) to prevent problems that could prevent certain addresses from being accessible.

Earlier, I mentioned that I/O devices could be interfaced with the 8051, and this can be done very simply by using one of P2's bits for the _CS or clock pin on the device. The last example 8051 application shows a memory mapped I/O solution using similar wiring. (See Fig. 2-12.)

For the example circuit in Fig. 2-12, writes to Device 1 could be carried out by writing to address 0FE00h, Device 2 can be read at 0FD00h, and the ROM (Device 0) could be read starting at 0FC00h. This scheme allows each device to be accessed without affecting any of the others or requiring extensive "glue" logic to determine whether or not reads or writes should be carried out. The lower 8 address bits are used for selecting registers or addresses within the peripheral devices.

One really interesting thing that can be done with the 8051 is to use one memory for *both* control store and RAM. This is done by ANDing _PSEN and _RD together (Fig. 2-13).

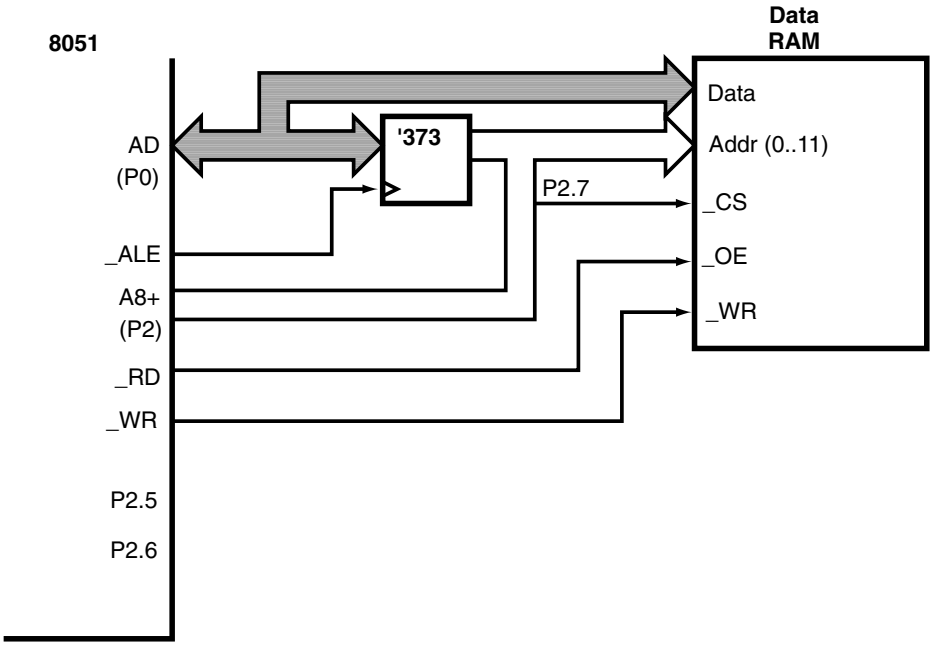

FIGURE 2-11 8051 2K external SRAM access.

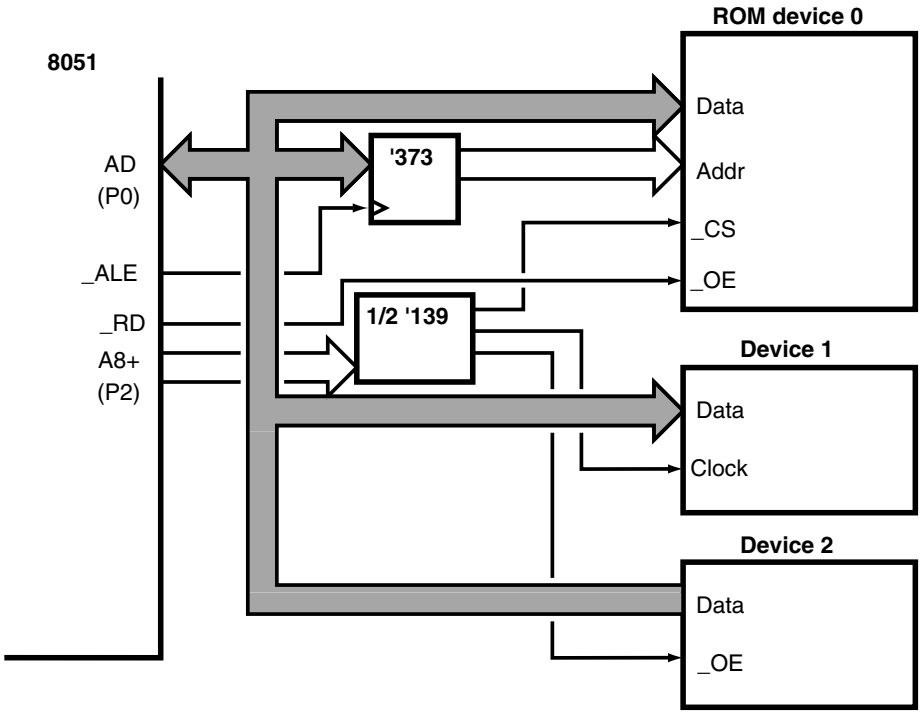

FIGURE 2-12 8051 external memory mapped I/O with a data ROM.

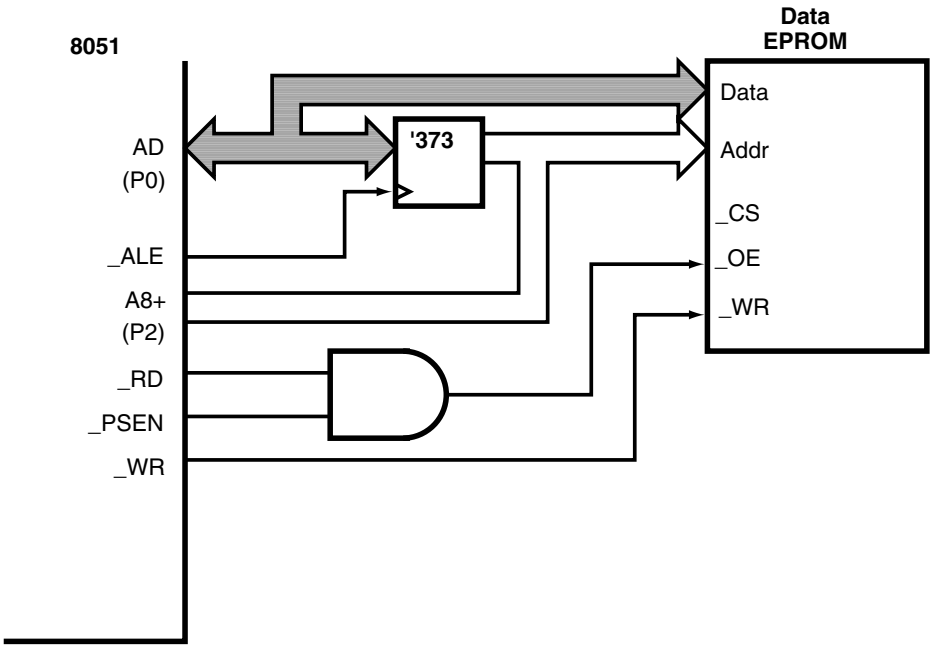

FIGURE 2-13 8051 shared control store/data SRAM.

By doing this, I've now created an 8051 application that can write to its control store (an immediate use would be for creating an 8051 experimental debug tool for downloading code into the SRAM). This circuit is used for the emulator presented later.

It also poses an interesting philosophical question: By creating this circuit, have I created a Von Neumann architected processor out a Harvard Architecture?

Interrupts

The "bottom" of the 8051's control store memory map can be represented as shown in Fig. 2-14. Address 0 is the reset (and watchdog timer overflow) execution vector address. When the 8051 is reset, execution jumps to this address. When an interrupt request is acknowledged, execution jumps to the appropriate address (i.e., interrupt 0 goes to address 03h, interrupt 1 goes to address 0Bh, and so on).

The exciting thing is, while other microcontrollers operate in a similar fashion, they don't provide memory space to allow an application to handle each individual interrupt directly in line. The 8 bytes of control store available for each interrupt handler can either be used to house the entire interrupt handler or jump to a more substantial handler.

I always look for features built into devices that make my life easier. By "easier," I'm trying to say that I have to do less thinking. Having 8 bytes available for instructions makes simple interrupt handlers (i.e., reset timer interrupt and increment a real-time clock value) very easy to implement.

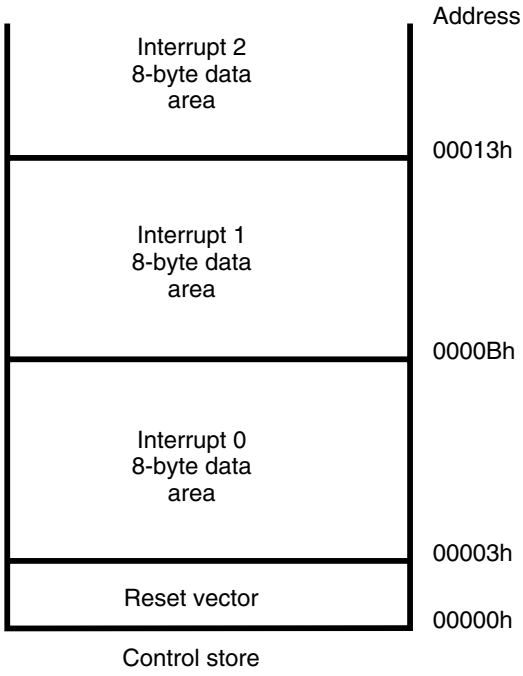

FIGURE 2-14 **8051 interrupt addresses.**

If you're thinking that the 8 bytes are probably not sufficient to save the context registers before handling the interrupt, you're probably correct if you're thinking about your experiences with other processors. However, because there is no zero flag in the 8051's PSW register, surprisingly complex operations can be carried out in 8-byte interrupt handlers. Later in the book, I will show you many examples of this.

An important feature that makes writing short interrupt handlers easier is the automatic interrupt controller hardware reset available for the basic 8051 interrupts.

8051 Instruction Execution

The 8051 processor core is designed differently than most other microcontrollers. The 8051 contains a microcoded processor that is in contrast to other microcontrollers that use a hardwired one. (See Fig. 2-15.)

What is a microcoded processor? This is really a processor within a processor, or a state machine that executes each different instruction as the address to a subroutine of instructions. When an instruction is loaded into the instruction-holding register, certain bits of the instruction are used to point to the start of the instruction routine (or microcode) and the microcode instruction decode-and-processor logic executes the microcode instructions until an instruction end is encountered.

As a quick aside, I should point out that having the instruction-holding register appear wider, in Fig. 2-15, than the control-store memory is not necessarily a mistake. The control store in the 8051 is only 8 bits wide, and quite a few of the instructions are more than 8 bits. This could mean that, before an instruction can be executed, the entire instruction might have to be loaded into the holding register (which can take additional time).

A hardwired processor uses the bit pattern of the instruction to access specific logic gates (possibly unique to the instruction), which are executed as a combinatorial circuit to carry out the instruction. (See Fig. 2-16.)

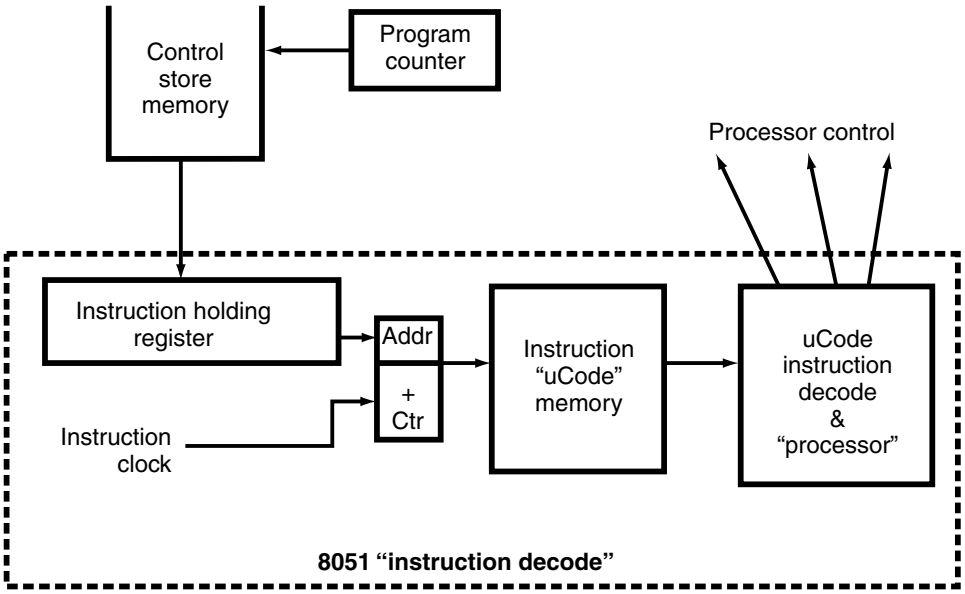

FIGURE 2-15 8051's microcoded processor.

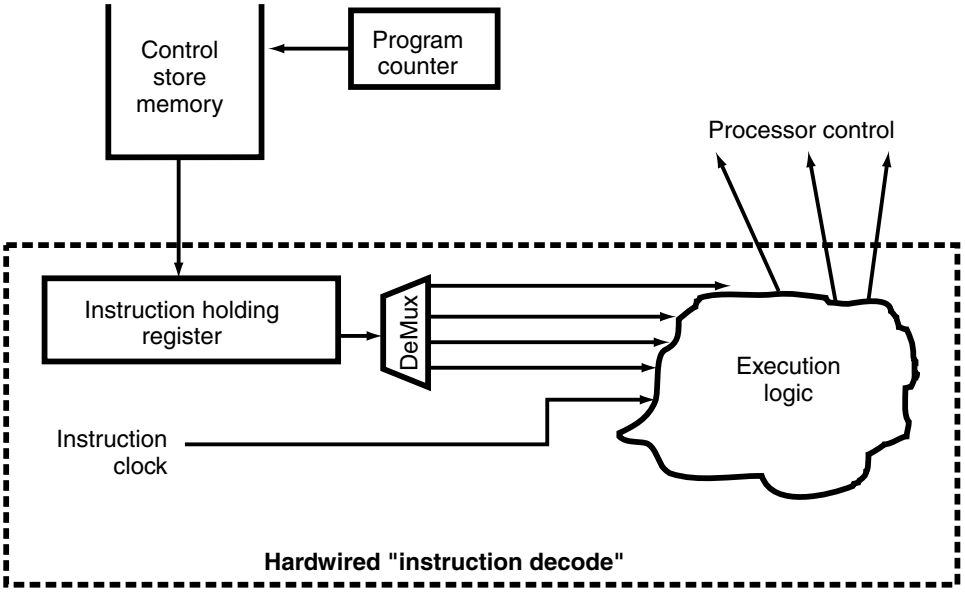

FIGURE 2-16 Hardwired instruction processor.

Each of the two methods offer advantages over the other. A microcoded process is usually simpler than a hardwired one to design, and the actual design can be implemented faster and with less chance of having problems at specific conditions.

A great example of the quick-and-easy changes that a microcoded processors allow was a number of years ago when IBM wanted to have a microprocessor that could run 370 assembly language instructions. Before IBM began to design their own microprocessor, they looked around at existing designs and noticed that the Motorola 68000 had the same hardware organization as the 370 (although the instructions were completely different). IBM ended up paying Motorola to rewrite the microcode for the 68000 and came up with a new microprocessor that was able to run 370 instructions, but at a small fraction of the cost of developing a new device.

A hard-wired processor is usually a lot more complex because the same functions have to be repeated over and over again in hardware (how many times do you think that a register read or write function has to be repeated for each type of instruction?). This means that the processor design will probably be harder to debug and be less flexible than a microcoded design, but it will probably run a lot faster.

The "true" (original) 8051 uses a microcoded processor that requires at 12 to 24 clock cycles for each instruction to execute. Later in this book, I will present three 8051-compatible devices that have redesigned processor cores aimed at improving the execution speed of the 8051's instruction set.