Chapter 3

The ARM Processor

To develop programs in a low level language it is necessary to select the type of CPU and know some features of the internal organisation. The CPU family selected here is that designed by ARM Holdings plc., and the internal information required is provided by the **programmer's model**, Figure 3.1. In a few cases where a CPU version must be specified ARM7TDMI is selected. Any system that includes an ARM core will also have some memory and the input and output, IO, systems required for the task.

3.1 THE CPU ARCHITECTURE

A complete system incorporating an ARM CPU, memory devices and the IO devices is built as outlined in Chapter 1. The assembly language programmer must know where temporary values are held in the CPU and the operations that can be performed using them. A major feature of any CPU is the block of registers whose contents the programmer can manipulate. **Registers** are similar to words of memory as each holds a fixed size binary pattern representing numbers, codes for letters, etc. They are used for values that are required immediately. Some registers are reserved for special tasks set by the CPU designer; the others are general purpose and are often called **scratch pad** registers. The action of an instruction, an **operation**, is to transfer values to and from the registers; often the values are modified during the transfer.

Figure 3.2 is a simplified outline of the internal organisation of the ARM7; it shows the paths along which data moves in the CPU and the connections for data transfer between the CPU and memory. All ARM registers are 32-bit ones, the CPU is said to have a 32-bit word length. The registers available to programmers are shown in the Programmers' Model, Figure 3.1. For initial studies the bank of sixteen registers r0 to r15 and the current program status register, CPSR, are the important ones and are shown without shading in Figure 3.1. The shaded registers are only used by more complex CPU operating modes which are not required for simple programming tasks.

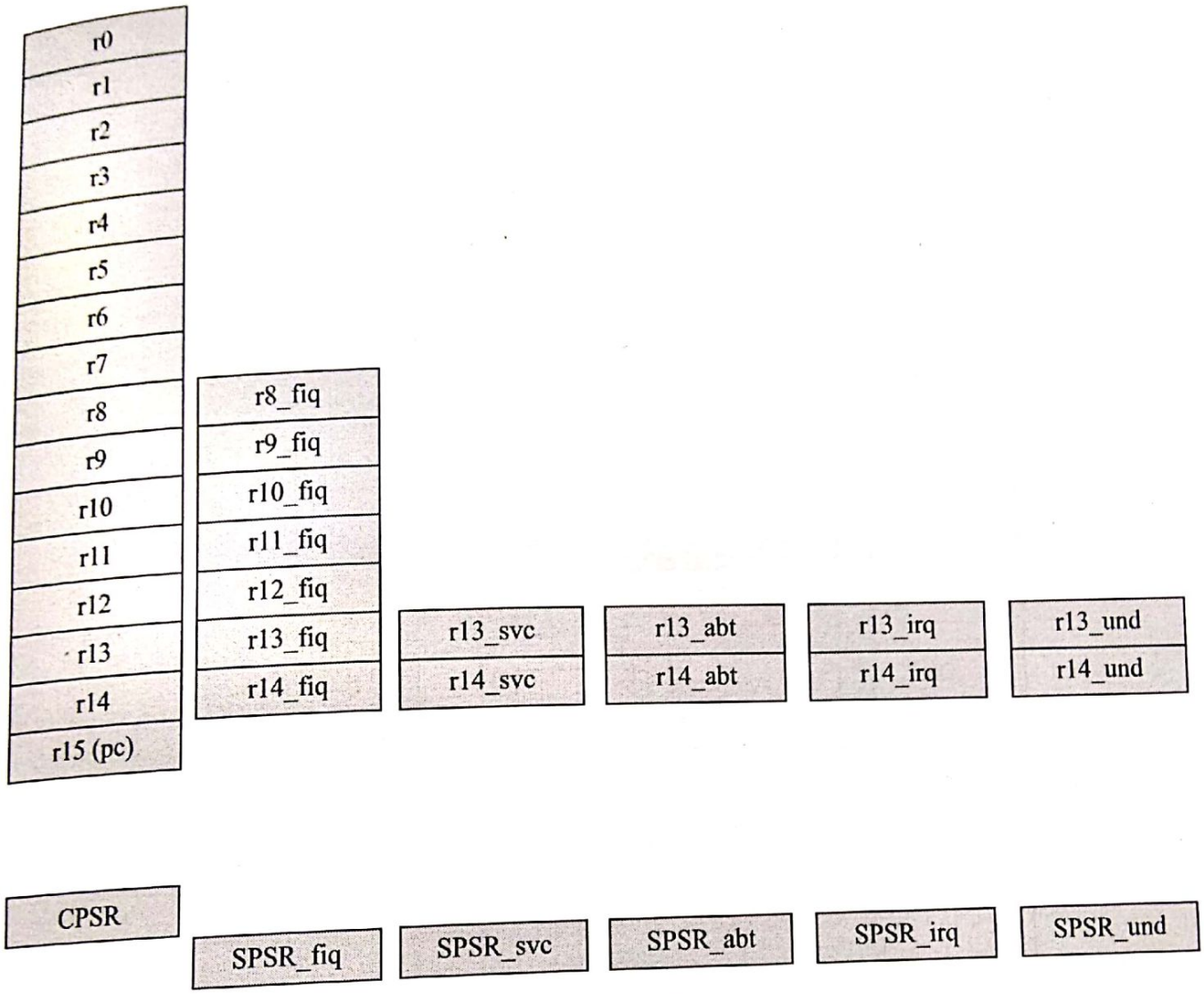


Figure 3.1 ARM Programmers' Model

When the CPU executes instructions it transfers values to and from registers. The programmer uses registers r0 to r11 as general purpose ones; that is r0 to r11 hold 32-bit values for any purpose the programmer chooses during execution of the program. Registers r13, r14 and r15 perform special functions and should only be used for these. ARM suggest additional restrictions in register use when a program consists of several modules. For advanced tasks r12 is used for a special purpose that is not examined. Although r12 may be used as a general purpose register when not used for the special purpose the risk of creating program faults is reduced if it is only used for the special function.

Register r15 is the most important CPU register and its alternative name is **program counter, pc**, because it holds the address of the next instruction to be fetched for execution. The program counter is automatically advanced after each instruction is fetched. Although it is possible, programs should not alter the value of the program counter as if it as a general purpose register; also programs should not directly use the contents of r15. There are exceptions to these rules but they are special well defined cases.

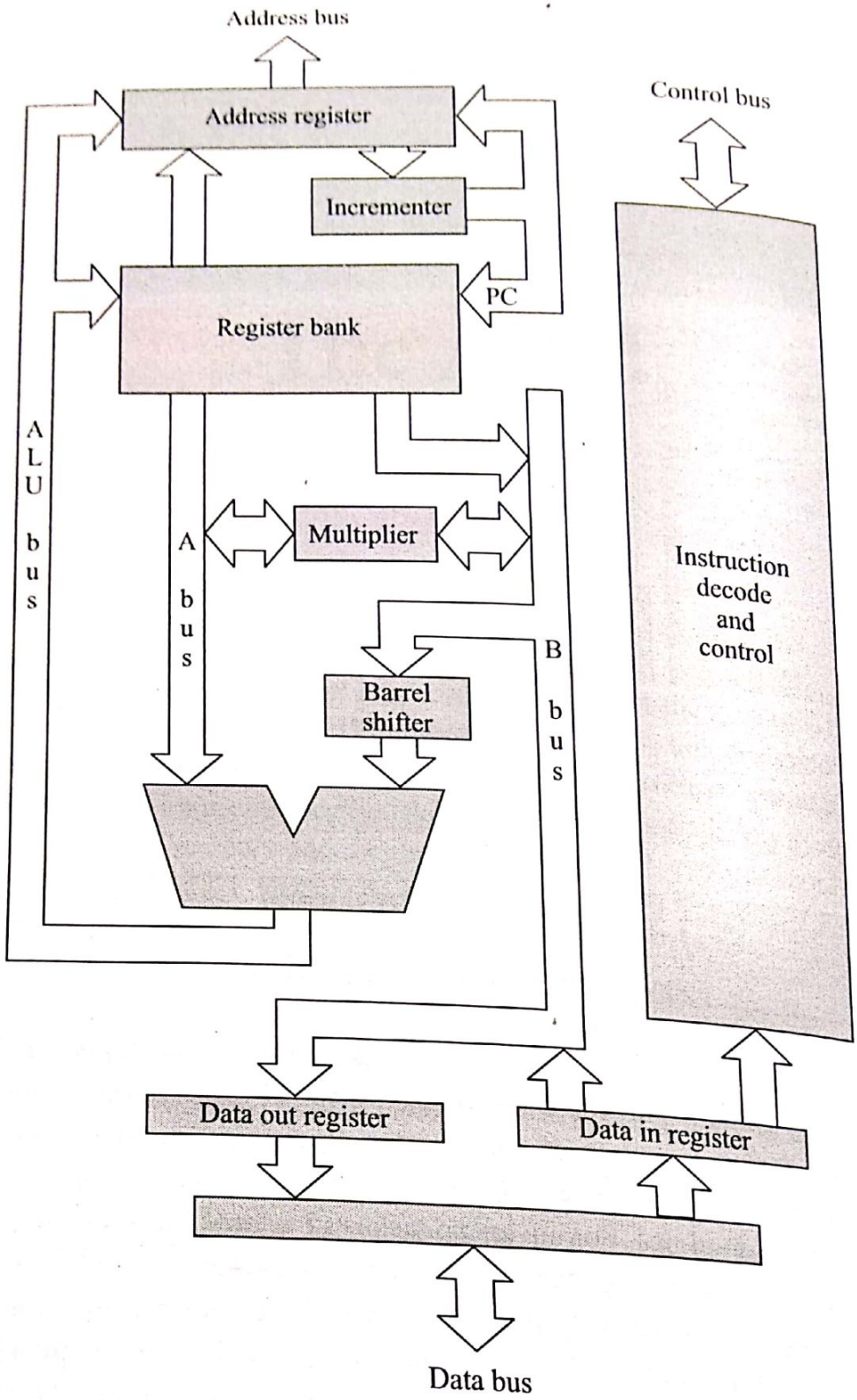


Figure 3.2 Simplified outline of the internal ARM7TDMI structure

Registers with special uses have alternative names; as indicated r15 is the program counter, p
Either r15 or pc may be used as the register name but pc is preferred as it indicates the registe
function; that is use pc rather than r15 to improve documentation. The von Neumann cycle, Section 1
described operation as a fetch-decode-execute cycle. To perform the fetch the CPU uses the addr

held in the program counter. The von Neumann cycle for an ARM processor is more fully described in Figure 3.3.

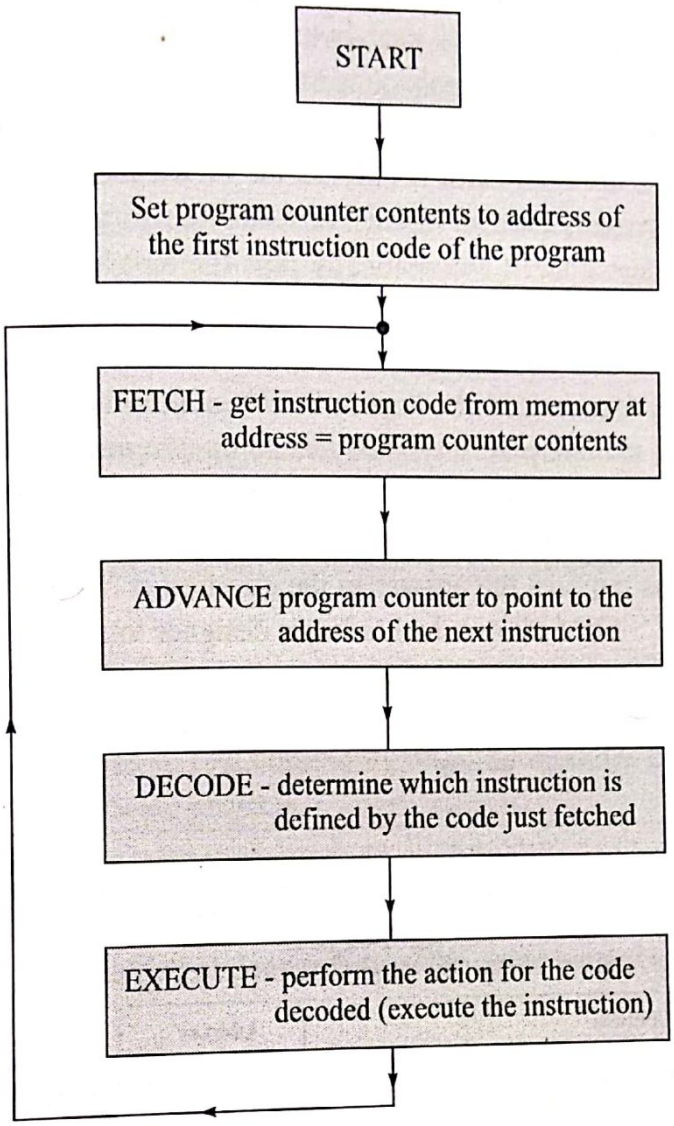


Figure 3.3 Detailed von Neumann cycle sequence

Here, and previously, the problem of *'How does it start?'* has been ignored. Processor systems include a reset circuit which forces the system into a known condition at power on and when a reset switch is pressed; the reset process forces the program counter to a specified value. Most programs are developed using tools that handle the start up process during testing, this allows the exact start up details to be ignored initially.

3.2 MEMORY FOR ARM SYSTEMS

Almost all ARM features are 32-bit; registers hold 32-bit values, instruction codes are 32-bit numbers, addresses are 32-bits, and most calculations are performed using 32-bit values to produce 32-bit results. As 32-bit values are used for addresses then 2^{32} memory locations can be used. 2^{32} is 4,294,967,296 which is a very large number that is usually written as 4G where G indicates giga. In the S.I. system giga represents 10^9; in digital electronics it is used for 2^{30} which is about 7% larger. Very few applications

will require the maximum amount of memory; the amount used depends on the application; very small systems have a few thousand words and large ones several million words.

Usually the manipulation of data by ARM programs uses 32-bit quantities, words. However the memory organization is such that every 8-bits, each byte, has a different address with 32-bit word values stored at four consecutive addresses. This rarely concerns the programmer provided sequences of 32-bit values are placed in memory **starting at addresses that are divisible by four;** that is the two LSBs of the address of the first byte are both zero. If only 32-bit values are used the CPU and development tools will automatically keep everything in order. For example every instruction is 32-bits; when an assembler or compiler produces code it automatically puts the first byte of the first instruction at an address divisible by four. As all instruction codes are four bytes the following codes are all correctly positioned. When the CPU fetches an instruction it fetches all 32-bits and adds four to the program counter. That is *'ADVANCE program counter to point to the address of the next instruction'* in the von Neumann cycle of Figure 3.3 is more precisely *'Add four to the program counter'*.

The feature, **byte addressing**, where each group of eight bits has a different address has many advantages and is used by many types of CPU. For some advanced tasks programmers must know how the memory is organised; this requires the answer to the question is *'How are 32-bit values arranged as four bytes in the memory?'* ARM allows the hardware designer to select either of two methods; the method is usually fixed when the CPU is built and cannot be changed by the programmer but systems can be built that allow selection by the programmer. The two methods are termed *'little-endian'* and *'big-endian'*.

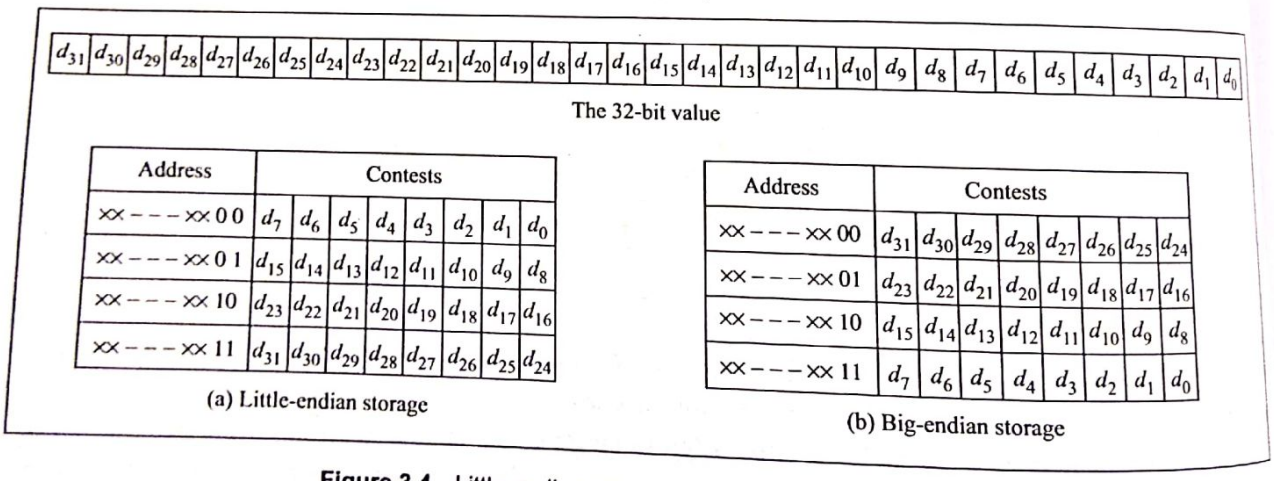


Figure 3.4 Little-endian and big-endian memory organisation

For little-endian systems the least significant 8-bits of the 32-bit quantity are at the lowest address of the four used with the LSB in the LSB position of the memory byte. The next 8-bits are in the memory at the next address, the third 8-bits are in the next location and the most significant 8-bits are in the last address, Figure 3.4a. Big-endian systems have the most significant 8-bits of the 32-bit quantity at the lowest address with the most significant bit, MSB, in the MSB position of the memory byte; the next most significant 8-bits are at the next address, and so on; Figure 3.4b. CPU designers argue about the best arrangement but the choice has little effect on system performance. Little-endian form will be assumed in the small number of cases for which it is necessary to know the details of number storage.

IO devices are connected to an ARM core as if they are memory components; ARM systems have memory mapped IO. Input devices behave as a memory that can only be read from and output devices as a memory that can only be written to. Many IO devices are 8-bit ones so the ability to address individual bytes is important for IO operations. A low level programmer often needs to know the addresses of IO devices and special restrictions on their use. Also systems rarely have memory at all addresses; the addresses used and the types of memory, ROM or RAM, at each address must be known when building complete programs.

Most tools do not require the addresses of the memory units connected to the CPU as part of the assembly language program. The code is produced in two stages, although a single stage is adequate for simple programs. Development tools are designed for creating large complex software divided into modules with each developed separately. The assembler processes each source module separately producing intermediate output for each one. A further tool, a **linker,** combines the intermediate modules into a single structure and determines the correct addresses. This two stage approach supports:

- Modular design of software.
- Many programmers working on a large software project simultaneously, each responsible for different modules.
- Development of mixed language programs; some modules in assembly language and others in one, or more, high level languages.
- Provision of library facilities so that commonly required modules can be developed once then used by many different software projects.
- Simple modification of programs so they run on slightly different hardware.

In simple situations the programmer is not aware of the two stage process; it is only when more complicated tasks are met that a detailed understanding of the development tool capability and operation may be necessary.

3.3 INSTRUCTION EXECUTION

As every ARM instruction is a 32-bit code number there are 4,294,967,296 possible instruction codes, some are not used but a very large number are used. The processor follows the von Neumann cycle with the contents of the program counter, register r15 or pc, used to indicate the memory addresses of instructions. The control circuits in the CPU use the address in the program counter and fetch, read, the contents of the memory at this address. Once the contents are in the CPU the value in the program counter is advanced by adding four to it. The 32-bit value fetched is decoded to determine which instruction it represents. The processor then performs the actions to execute the instruction; the actions are fully defined by the code number.

ARM is a RISC processor with a **pipelined operation**; while one instruction is being executed the one to be executed next is being decoded and the one after that is being fetched as in Figure 3.5. The operation is more complicated than the von Neumann cycle of Figure 3.3; the exact sequence in which instructions are executed is identical but the rate at which they are executed is much higher. A pipelined CPU simultaneously performs fetch, decode and execute operations and when an instruction is executed the program counter value is not always that expected. As Figure 3.5 does not apply to all instructions

the value is not easy to predict; for example some instructions require two time periods for execution. Figure 3.5 is for ARM7; later versions are more complicated. Furber [1] describes the operation of the various forms of pipeline with reasons for the changes. The pipeline is one reason why the assembly language programmer should avoid changing the contents of the program counter, or even use the value it holds, except in certain well defined situations.

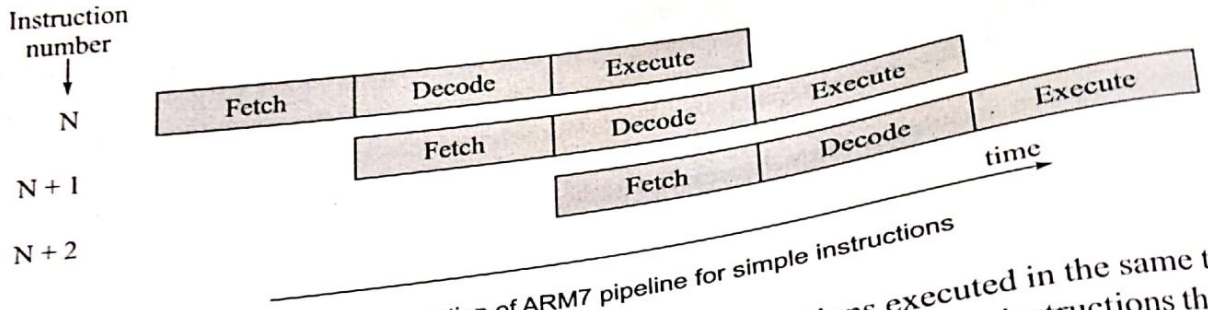


Figure 3.5 Operation of ARM7 pipeline for simple instructions

The pipelined structure of the ARM processor with most instructions executed in the same time that it takes to fetch or decode them means that the ARM instruction set can only have instructions that can be executed very quickly. Consequently only instructions which are simple to execute are possible; only a small number of different instructions meet the requirement of fast execution hence CPUs designed with this form have only a small number of instructions. This is the origin of the name reduced instruction set computer, RISC; the primary design aim is fast instruction execution rather than the use of a small instruction set.

3.4 DEVELOPMENT TOOLS

ARM processors are supported by powerful development tool packages from ARM Holdings plc., and by alternatives from other suppliers. One alternative set is that from the Free Software Foundation, usually called the GNU software tools, Appendix F. In general development tools are complex large computer packages that perform many functions. They are intended for development of complete software systems for large commercial applications and the large number of features included makes using them difficult for the beginner. Except in Appendix F it is assumed that either the ADS or RealView assembler from ARM is used; features only available in RealView are not used so that examples work with either version. There are differences in the syntax, text rules, for assembly language programs when packages from different suppliers are used; the main differences using the GNU tools instead of ARM tools are outlined in Appendix F.

Evaluation versions of the ARM tools are available; they can be downloaded from the internet site of Keil Elektonik GmbH which became an ARM subsidiary in 2005 (the evaluation tools are time limited but repeat downloading is allowed). The Keil site allows download of both the RealView and GNU assemblers. Most versions have no time limit but the maximum program size using the evaluation version of the RealView tools is 16kbytes of code; this is adequate for initial studies of assembly language.

Any development package that includes some simple set-up instructions, an assembler and support for testing programs may be used provided any syntax differences are known and understood. Most packages run on standard desktop computers and are available for machines running either the Windows or Unix/Linux operating systems. The assembler in the packages is called a **cross-assembler** because it runs on one type of CPU, the one in the desktop computer, but produces code for another type of CPU, in this case the ARM core.

Having designed a program structure and composed the assembly language program the text is input to a **source file** using a text editor; usually one is included in the development system. A word processor **must not** be used to create source files although many text editors allow text to be cut and pasted from a word processor package. Once the source file is complete the assembler is used to produce the binary code which is tested to ensure that it performs as specified. Thorough testing covering all possible situations is an essential part of software development. First tests are often performed using a simulator provided with the development system. A simulator is a program that runs on the same computer as the development tools and imitates the actions of the target CPU running the code produced. Simulators include aids that allow the user to inspect the register contents as the program runs. The Keil evaluation package includes a powerful simulator with provision for simulating input from, and output to, external devices by the ARM program.

Most development tools provide simple methods of loading programs into hardware target systems that contain a real CPU and monitoring their execution when running on this CPU. Several low cost simple development targets are available for ARM systems although some are restricted to using the GNU tools.

3.5 SUMMARY – WHAT THE PROGRAMMER NEEDS TO KNOW

The programmer should consider the ARM CPU as operating on a bank of sixteen registers, r0 to r15. For most programs r0 to r11 are general purpose and may be used however the programmer chooses; r13, r14 and r15 should be reserved for their special functions that will be introduced as required. r12 may be used as a general purpose register by simple programs but is reserved for special use in advanced programs.

System memory is a block of 32-bit storage units at contiguous memory addresses in a range set by all possible 32-bit numbers; individual addresses apply to 8-bit values. The 8-bit values are grouped so that values from four adjacent addresses starting at the address with the two least significant bits zero form the 32-bit word value.

This brief outline description provides enough detail of the CPU structure to begin studying the process of preparing ARM programs in assembly language. As programming can only be learned by performing exercises development software and testing tools are required. Evaluation packages are adequate for exercises but are too restricted for developing commercial applications. A good knowledge of how to use the development package chosen must be acquired.

THE MICROCHIP PIC MCU PROCESSOR ARCHITECTURE

When starting to research a new device, whether it is a microcontroller such as one of the many PIC® MCU part numbers or a simple logic gate, one of the first things that you have for reference information is the block diagram. A well-drawn block diagram, such as the one taken from the PIC16C61 microcontroller datasheet in Fig. 6.1, actually has all the information that you need to understand and start working with the chip. Unfortunately, block diagrams can be intimidating and confusing when you look at them for the first time. I dare say that many people will skip right over them without spending any time trying to understand how data flows and what features are available on the chip, and this is unfortunate because there is a wealth of information that will help you to visually understand and remember how the chip works and allow you to follow the flow of data through the chip.

The purpose of this chapter is to help you to understand how programs execute and manipulate data in the PIC microcontroller's processor. To do this, I am going to rely on the block diagram that is available in each of the PIC MCU datasheets. I have found that this is a very useful way of going about the task of learning about the processor and how it works because the diagram is an architectural drawing of its inner workings. When you understand the block diagram, you will understand how each instruction executes, and this will give you some insights into how to structure your assembly-language programs so that they are as efficient as possible.

The processor block diagrams are very similar for each of the PIC microcontroller processor families (consisting of the low-end, mid-range, PIC17, and PIC18 architectures), and for this reason, I will explain the mid-range devices in detail and then review the architectural differences in the other PIC MCU families. These differences mainly center on how data is accessed in different register banks, how intrapage jumps and calls are executed, and how data is indexed and stored in stacks. Even without these sections explaining the differences in the mid-range PIC microcontroller architecture, you should be able to understand how these processors work by reviewing the block diagram at the start of the datasheets.

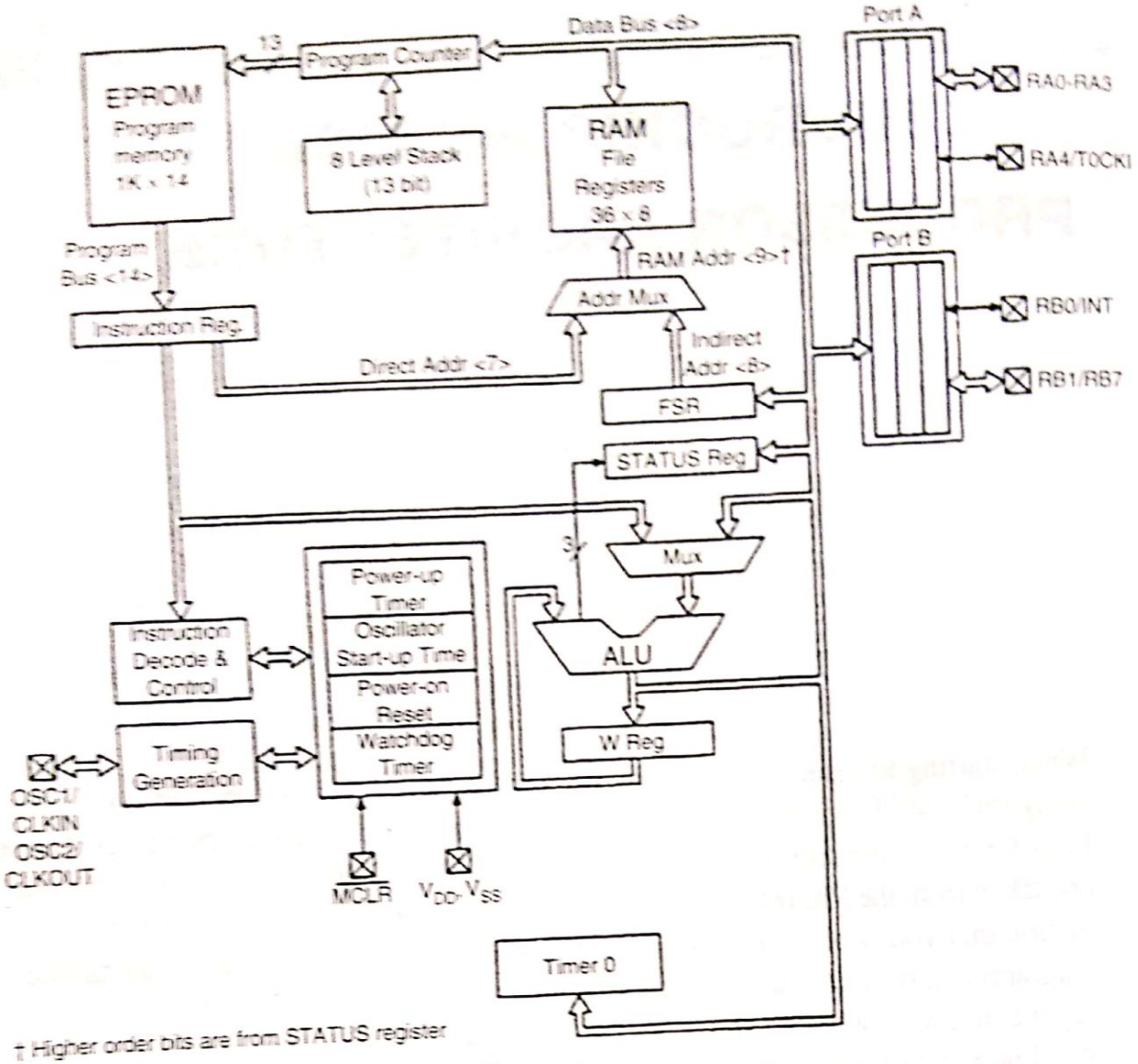


Figure 6.1 PIC16C61 block diagram.

The CPU

In the microchip datasheets you will find that the PIC microcontroller's processor is described as a "RISC-like architecture . . . separate instruction and data memory (Harvard architecture)." In this chapter I want to explain what this means for people who do not have Ph.D.s in computer architectures, as well as help explain how application code executes in the PIC MCU processor. The processor may seem to be very complex and different from other devices you've worked with before, but I believe that it is very intelligently designed and works in a very logical manner. Despite the complex written description of the processor, you will discover that it is actually quite straightforward and designed to simplify the implementation of many complex applications and programming algorithms.

The PIC microcontroller processor can be thought of as being built around the *arithmetic/logic unit* (ALU), which provides basic arithmetic and bitwise operations for the processor. There are a number of specific-use registers that control operation of the CPU as well as input/output (I/O) registers and data-storage (RAM) registers. In this book I call the

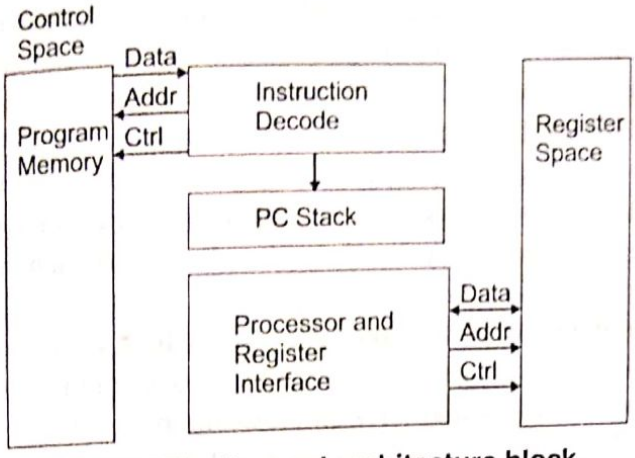


Figure 6.2 Harvard architecture block diagram.

specific-use registers *hardware registers* or *I/O registers* depending on the function they perform. The hardware registers also allow direct manipulation of functions that usually are invisible to the programmer, such as the program counter, to allow for advanced program functions. Data-storage (RAM or variable) registers are called *file registers* by Microchip.

The registers are completely separate from the program memory and are said to be in their own "spaces." This is known as *Harvard architecture* and is shown in Fig. 6.2. In the figure, note that the program memory and the hardware to which it is connected are completely separate from the register space. This allows program memory reads for instructions to take place while the processor is accessing data and processing it. This capability allows the PIC microcontroller to execute software faster than many of its contemporaries.

Instruction execution takes place over four clock cycles, as shown in Fig. 6.3. During an instruction execution cycle, the next instruction to be executed is fetched from program memory. When the next instruction is executing, the processor is fetching the next instruction after it. After an instruction has been fetched and is latched in a

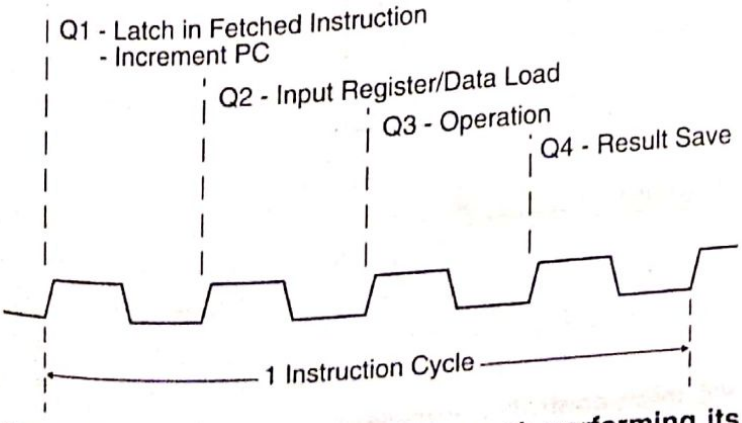


Figure 6.3 Four clock cycles, each performing its own task, make up a single instruction cycle.

holding/decode register, the program counter (used to address which instruction is being executed) is incremented. This is known as Q1. Next (Q2), data to be processed (often with the data in the accumulator or "working" register, which will be described below) is read and put into temporary buffers. During Q3, the data-processing operation takes place. Finally, the resulting data value is stored during Q4, after which the process repeats itself for the next instruction (which is put into the holding register while the current instruction is executing). These four cycles that take place with each "tick" of the clock are known collectively as an *instruction cycle*.

Since the instruction cycle is made up of the four Q cycles, which is equivalent to four clock cycles, the instruction execution speed is said to be one-quarter the clock speed. For example, an application that has a 4-MHz clock would be running 1 million instruction cycles per second (MIPS). In the PIC18 processors, there is a built-in phased-locked loop circuitry that multiplies the external clock's speed four times. This means that for PIC18 chips with the phased-locked loop active, the instruction cycle is equal to the chip's clock.

There are three primary methods of accessing data in the PIC microcontroller. *Direct* addressing means that the register address within the register bank (explained below) is specified in the instruction. If a constant is going to be specified, then it is specified *immediately* in the instruction. The last method of addressing is to use an *index* register that points to the address of the register to be accessed. *Indexed addressing* is used because the address to be accessed can be changed arithmetically. In other processors, there are additional methods of addressing data, but in the PIC microcontroller, these are the only three.

When accessing registers in the mid-range PIC microcontrollers directly, 7 address bits are explicitly defined as part of the instruction. These 7 bits result in the ability to specify up to 128 addresses in an instruction, as shown in Fig. 6.4.

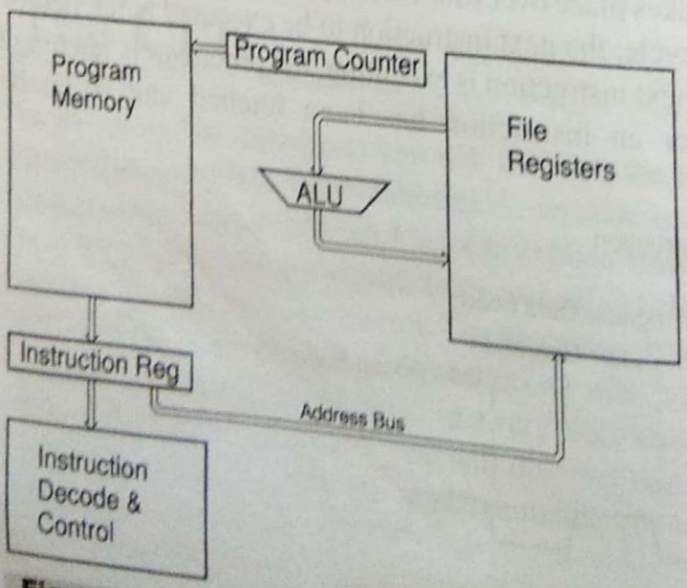


Figure 6.4 Basic PIC microcontroller processor architecture.

These 128 register addresses are known as a *bank*. To expand the register space beyond 128 addresses for hardware and variable registers, Microchip has added the capability of accessing multiple banks of registers, each capable of registering 128 addresses in the mid-range PIC microcontrollers. The low-end PIC microcontrollers can access 32 registers per bank, also with the opportunity of having four banks accessible by processor for up to 128 register addresses in total. This will be explained later in this chapter, along with how register addressing is implemented for the PIC17C and PIC18C processors.

The ALU shown in Fig. 6.4 is an acronym for the *arithmetic/logic unit*. This circuit is responsible for doing all the arithmetic and bitwise operations, as well as the conditional instruction skips implemented in the PIC microcontroller's instruction set. Every microprocessor available today has an ALU that integrates these functions into one block of circuits. The ALU will be discussed later in this chapter.

The *program counter* maintains the current program instruction address in the *program memory* (which contains the instructions for the PIC microcontroller processor, each one of which is read out in sequence and stored in the *instruction reg* and then decoded by the *instruction decode and control* circuitry.

The program memory contains the code that is executed as the PIC microcontroller application. The contents of the program memory consist of the full instruction at each address (which is 12 bits for the low-end, 14 bits for the mid-range and 16 bits for both the PIC17 and PIC18 devices). This differs from many other microcontrollers in which the program memory is only 8 bits wide, and instructions that are larger than 8 bits are read in subsequent reads. Providing the full instruction in program memory and reading it at the same time result in the PIC microcontroller being somewhat faster in instruction fetches than other microcontrollers.

The block diagram in Fig. 6.4, while having 80 percent or more of the circuits needed for the PIC microcontroller's processor is not a viable processor design in itself. As drawn in Fig. 6.4, there is no way to pass data to the program memory for immediate addressing, and there is no way to modify the program counter. As I work through this chapter, I will be fleshing out Fig. 6.4 until it is a complete processor that can execute PIC microcontroller instructions.

To implement two-argument operations, a temporary holding register, often known as an *accumulator,* is required to save a temporary value while the instruction fetches data from another register or is passed a constant value from the instruction. In the PIC microcontroller, the accumulator is known as the *working register* or, more commonly, as the *w register.* The w register really cannot be accessed directly as a register address in itself in the low-end and mid-range PIC microcontrollers. Instead, the contents must be moved to other registers that can be accessed directly. The w register can be accessed as an addressed register in the PIC17 and PIC18 devices. Every arithmetic operation that takes place in the PIC microcontroller uses the w register. If you want to add the contents of two registers together, you would first move the contents of one register into the w register and then add the contents of the second to it.

The PIC microcontroller architecture is very powerful from the perspective that the result of this operation can be stored either in the w register or the source of the data.

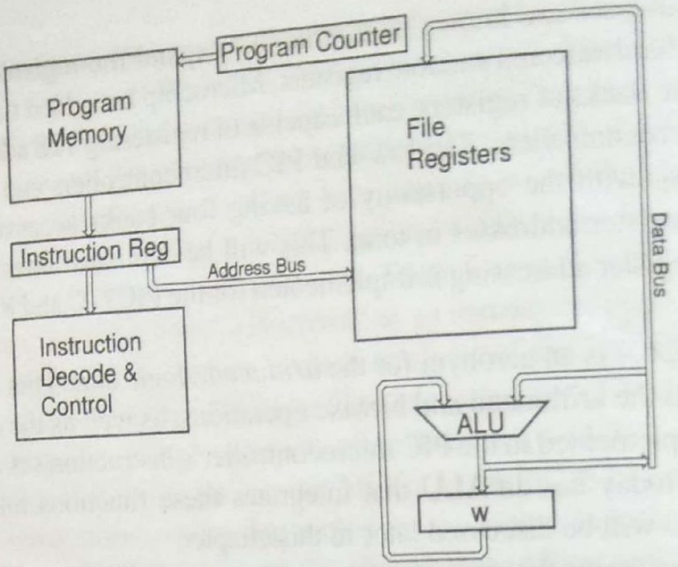


Figure 6.5 PIC microcontroller processor architecture with the w register and file registers as source and destination for ALU operations.

Storing the result back into the source effectively eliminates the need for an additional instruction for saving the result of the operation. There is a lot of flexibility in how instructions are executed to provide arithmetic and bitwise operations for an application.

Adding the w register changes how the ALU is wired in the PIC microcontroller processor block diagram, as shown in Fig. 6.5. Note that the ALU has changed to a device with two inputs (which is the case in the actual PIC microcontroller's ALU) and that the contents of the w register are used as one of the inputs. You also should note that when a result is passed from the ALU, it could either be stored into the w register or in one of the file registers. This is a bit of foreshadowing of one of the most important features of the PIC microcontroller architecture and how instructions execute.

Figure 6.5 shows the PIC microcontroller at its simplest level. This simple circuit can execute well over half the PIC microcontroller's instructions.

Hardware and File Registers

If you have worked with other processors and computer systems, you probably will be surprised by the close coupling and shared memory space of the PIC microcontroller's processor's registers, hardware I/O registers, and variable RAM. This is a result of the small (5-bit addressing for low-end devices and 7-bit addressing for mid-range devices) register space accessible to the processors. Despite being somewhat unusual, this close coupling of registers for both variable storage and hardware I/O registers provides you with a common means of accessing, processing, and updating the contents of registers regardless of their function, using a single set of tools.

In the mid-range PIC microcontroller, each instruction that accesses a register contains the addresses within the given bank with a maximum bank size of 7 bits, which allows up to 128 different addresses. In each bank, the registers fall within four distinct groups

TABLE 6.1 BASE REGISTER ADDRESSES BY PIC MICROCONTROLLER ARCHITECTURE FAMILY

REGISTER	LOW-END	MID-RANGE	PIC17	PIC18
WREG	Not accessible	Not accessible	0x0A	0xFE8
STATUS	0x03	0x03	0x04/0x06	0xFD8
PCL	0x02	0x02	0x02	0xFF9
PCLATH	Page bits in STATUS	0x0A	0x0e	0xFFB/0xFFA
FSR	0x04	0x04	0x01/0x09	0xFEA-0xFE9
INDF	0x00	0x00	0x00/0x08	0xFEF

- Processor registers
- I/O hardware registers
- Variable memory
- Shared or "shadowed" variable memory

The processor registers consist of STATUS, PCL, PCLATH (from mid-range devices), FSR, INDIF, and WREG (for high-end devices). These registers are always at the same addresses within the different PIC microcontroller families. These addresses are listed in Table 6.1. These registers can be accessed from within any of the register banks.

The I/O hardware registers consist of the OPTION, TMRO, PORT, I/O PINS and enable registers, INTCON, and other interrupt control and flag registers, along with any other hardware features built into the particular PIC microcontroller. The important difference between these registers and processor registers is that except for INTCON, these registers are bank-specific, and while some conventions are used for the placement of these functions, for part numbers, and for specific functions, the registers are located in different addresses. The registers with conventions are listed in Table 6.2.

As time goes on and more features become standard, you'll probably see the mid-range PIC microcontrollers standardize on a 32-byte processor and I/O hardware register block (also known as the *special function registers*, or SFRs) at the start of each bank.

Above the processor and I/O hardware registers, are the *file registers*, or variable memory. This memory can be bank-specific or shared between banks. In all PIC microcontrollers, there are a number of bytes that are always available (shared, or what I call *shadowed*) across all the register banks. This memory is used to pass data between the banks or, as I prefer to use them, to provide a common variable for sharing context register data during interrupts without having to change the bank specification in the status register. The shared memory is PIC microcontroller part number–specific and can be common across all banks or pairs of banks.

In the low-end PIC microcontrollers, many devices have multiple banks, but these multiple banks are strictly for providing additional file registers. Normally in these

TABLE 6.2 I/O REGISTER ADDRESSES BY PIC MICROCONTROLLER ARCHITECTURE FAMILY

REGISTER	LOW-END	MID-RANGE	PIC17	PIC18
OPTION	Uses OPTION Instruction	0x81	0x05	0xFD0
TMR0	0x01	0x01	0x0B/0x0C	0xFD7/0xFD6
PORTC-PORTA	0x07–0x05	0x07–0x05	Varies by part number	0xF82–0xF80
TRISC-TRISA	Uses TRIS port Instruction	0x87–0x85	Varies by part number	0xFD4–0xFD2
PORTD/TRISD	Not available	0x08/0x88	Varies	0xF83/0xFD5
PORTE/TRISE	Not available	0x09/0x89	Varies	0xF84/0xFD6
INTCON	Not available	0x0B	0x07	0xFF2
OSCCAL	0x05	Varies by part number	Not available	Varies by part number

PIC MCUs, the first 16 addresses of each bank (address 0 to 0x00F) are common, with the upper 16 bytes of each bank having file registers that are specific to them.

BANK ADDRESSING

One of the most difficult concepts for most people to understand when they first start working with PIC microcontrollers is the register banks used in the different PIC microcontroller architectures. The number of registers available for direct addressing in the PIC microcontroller is limited to the number of address bits in the instruction that are devoted to specifying register access. In low-end PIC microcontrollers there are only 5 bits (for a total of 32 registers per bank), whereas in mid-range PIC microcontrollers there are 7 bits available for a total of 128 registers per bank. The PIC18 can access 256 register addresses, but each bank is 128 registers in size.

In order to provide additional register addresses, Microchip has introduced the concept of *banks* for the registers. Each bank consists of an address space consisting of the maximum size allowable by the number of bits provided for the address. When a mid-range application is executing, it is executing out of a specific bank, with the 128 registers devoted to the bank directly accessible.

In each PIC microcontroller, a number of common hardware registers are available across all the banks. For mid-range devices, these registers are INDF and FSR, STATUS, INTCON (presented later), PCL, and PCLATH (also discussed later). These registers can be accessed regardless of the bank that has been selected. Other hardware registers may be common across all or some of the banks as well. In all mid-range PIC microcontrollers there are common file registers that are common across banks to allow data to be transferred across them.

Bank 0	Bank 1
Addr - Reg	Addr - Reg
00 - INDF	80 - INDF
01 - TMRO	81 - OPTION
02 - PCL	82 - PCL
03 - STATUS	83 - STATUS
04 - FSR	84 - FSR
05 - PORTA	85 - TRISA
06 - PORTB	86 - TRISB
07 -	87 -
08 - EEDATA	88 - EECON1
09 - EEADR	89 - EECON2
0A - PCLATH	8A - PCLATH
0B - INTCON	8B - INTCON
0C - 4F Shared - File Regs	8C - CF Shared - File Regs
50 - 7F - Unused	D0 - FF - Unused

Shaded areas indicate unused registers
- 0x000 Returned when these registers are read

Figure 6.6 PIC16F84 register map.

In Fig. 6.6, the PIC16C84's register space is shown for bank 0 and bank 1. When execution has selected bank 0, the PORTA and PORTB registers can be addressed directly. When bank 1 is selected, the TRISA and TRISB registers are accessed at the same address as PORTA and PORTB when bank 0 is selected.

To change the current bank out of which the mid-range application is executing, the RPx bits of the STATUS register are changed. To change between bank 0 and bank 1 or bank 2 and bank 3, RP0 is modified. Another way of looking at RP0 is that it selects between odd and even banks. RP1 selects between the upper (bank 2 and bank 3) and lower (bank 0 and bank 1) bank pairs. For most of the basic mid-range PIC microcontroller applications presented in this book, you will only be concerned with bank 0 and bank 1 and RP0.

At the risk of getting ahead of myself, the TRIS registers are used to specify the input or output operation of the I/O port bits. When one of the TRIS register bits is set, the corresponding PORT bit is in *input mode*. When the TRIS bit is reset, then the PORT bit is in *output mode*. To access the PORT bits, bank 0 must be selected, and to access the TRIS bits, bank 1 must be selected.

For example, to set PORTB bit 0 as an output and load it with a 1, the PIC microcontroller code would execute as

```
PORTB.Bit0 = 1;                  // Load PORTB.Bit0 with a "1"
STATUS.RP0 = 1;                  // Start Executing out of
Bank 1
TRISB.Bit0 = 0;                  // Make PORTB.Bit0 Output
STATUS.RP0 = 0;                  // Resume Execution in Bank 0
```

Microchip specifies that bank 1 registers are defined with the same address as bank 0 registers but with bit 7 set in their address specification. This means that for the mid-range PIC microcontrollers, bank 0 register addresses are in the range of 0 to 0x7F, whereas

bank 1 register addresses are in the range of 0x80 to 0xFF. Once the RP0 bit is set to select the appropriate bank, the least significant 7 bits of the address are used to access a specific register. This can be very confusing—the reason for having this specification is the FSR (index pointer) register, which is 8 bits in size. The FSR can access registers in both banks transparently. The Microchip TRISB register has the address value 0x86, which has bit 7 set and is in bank 1. PORTB has an address value of 0x006 and can only be accessed when bank 0 is selected.

When you start working with more complex mid-range PIC microcontrollers, which use all four banks, you will see registers with address bit 8 set, which indicates that the registers are in banks 2 and 3. These registers are accessed directly using the RP1 bit (along with RP0), and the least significant 7 bits of the Microchip-specified address are used as the address.

Specifying an address with bit 7 (or 8) set will result in the following message:

```
Register in operand not in bank 0. Ensure that bank bits are correct.
```

This indicates that an invalid register address has been specified and to make sure that execution is in the correct bits. Most people clear bits 7 and 8 of the defined register address to avoid this message. This can be done by simply ANDing the address with 0x7F to clear bit 7, but a somewhat more sophisticated operation normally is performed on the address to make sure that the register is accessed from the correct bank. Instead of ANDing with 0x7F to clear bit 7 for bank 1, the address is XORed with 0x80. By doing this, if the register is supposed to be in bank 1 (bit 7 of the address is set), then it will be cleared. If the register can only be accessed in bank 0 (bit 7 of the address is reset), then this operation will result in bit 7 being set and will cause the preceding message to be given. This is a nice way to ensure that you are not accessing registers that are not in the currently selected bank.

Using the XOR operation, the preceding example becomes

```
PORTB.Bit0 = 1;                    // Load PORTB.Bit0 with a "1"
STATUS.RP0 = 1;                    // Start Executing out of
                                   // Start Bank 1
(TRISB ^ 0x080).Bit0 = 0;          // Make PORTB.Bit0 Output
STATUS.RP0 = 0;                    // Resume Execution in Bank 0
```

This is also true for banks 2 and 3, which have address bit 8 set. In Table 6.3 I have listed the value of the XOR registers for specific banks. If the error message comes out of the register access, then you will know that you are accessing a register in the wrong bank. Note that the INDF, PCL, STATUS, FSR, PCLATH, and INTCON registers are common across all the banks and do not have to have their addresses XORed with a constant value to be accessed correctly.

Direct bank addressing is a very confusing concept and, unfortunately, very important to PIC microcontroller application development. I realize that it probably will be difficult for you to understand exactly what I am saying here, but it will become clear as you work through the example application code.

TABLE 6.3 BANK ADDRESS TO "RPX" BIT SETTINGS

BANK	RP1	RP0	ADDRESS RANGE	XOR VALUE
0	0	0	0x0–0x7F	None
1	0	1	0x80–0xFF	0x80
2	1	0	0x100–0x17F	0x100
3	1	1	0x180–0x1FF	0x180

The index register (FSR), as I indicated earlier, is 8 bits in size, and its bit 7 is used to select between the odd and even banks (bank 0 and bank 2 versus bank 1 and bank 3). Put another way, if bit 7 of the FSR is set, then the register being pointed to is in the odd register bank. This straddling of the banks makes it very easy to access different banks without changing the RP0 bit. For the preceding example, if I were to use the FSR register to point to TRISB instead of accessing it directly, I could use the code

```
PORTB.Bit0 = 1;                 // Load PORTB.Bit0 with a "1"
FSR = TRISB;                    // FSR Points to TRISB
INDF.Bit0 = 0;                  // Make PORTB.Bit0 Output
```

This ability of the mid-range FSR register to access both banks 0 and 1 is why I recommend that for many applications array variables should be placed in odd banks, and single-element variables should be placed in even banks. Of course, this is only possible if the entire file register range is not "shadowed" across the banks as in the PIC16F84 and other simple mid-range PIC microcontrollers that are used in introductory applications.

To select between the high and low banks with the FSR, the IRP bit of the STATUS register is used. This bit is analogous to the RP1 bit for direct addressing. Having separate bits for selecting between the high and low bank pairs means that data can be transferred between banks using direct and index addressing without having to change the bank-select bits for either case.

There is one thing that I have to note with regard to the FSR register and indirect addressing. Even though the FSR register can access 256 different register addresses across two banks, it *cannot* be used to access more than 128 file registers contiguously (or all in a row). The reason for this is the control registers contained at the first few addresses of each bank. If you try to wrap around a 128-byte bank, you will corrupt the PIC microcontroller's control registers with disastrous results.

ZERO REGISTERS

I don't really know if this qualifies as a feature, but unused registers in a PIC microcontroller's register map will return 0 (0x00) when they are read. This capability can be useful in some applications. *Zero registers* (undefined registers that return 0 when

read) are normally defined in the Microchip documentation as *shaded* addresses in the device register map documentation.

In Fig. 6.6, the PIC16F84's register map is shown with addresses 7 (PORTC registers) in each bank shaded, indicating that they return 0 when read. Of course, when these registers are written to, their values are lost and not stored in the register. (One might say the information has gone to the "great bit bucket in the sky.")

I am hesitant to recommend using the zero registers when programming. It is important to note that in different PIC microcontrollers, the zero registers are at different locations. Because of this, if code is transferred directly from one application to another and the zero register chosen is not available in the PIC MCU destination (e.g., a valid file or hardware register is at this location), then the code will not work correctly. Instead of using a hardware zero register, I would recommend that a file register be defined and cleared for the purpose of always returning 0.

The PIC Microcontroller's ALU

2.1 PIC18F MICROCONTROLLER FAMILIES

PIC microcontrollers are designed using the Harvard architecture (as explained in Chapter 1) that includes a separate memory for data and for programs (instructions). Figure 2-1 shows a simplified block diagram of a typical PIC microcontroller that includes five blocks: microprocessor unit (MPU), program memory, data memory, I/O ports, and support devices. There are many versions of PIC18F microcontrollers, ranging in package size from 18 pins to 100 pins, and their clock frequencies range from 25 MHz to 48 MHz. The primary differences between these versions are in available memory and I/O ports. The program memory varies from 4 KB to 128 KB, data memory from 256 bytes to 3968 bytes, data EEPROM from 128 bytes to 1 KB (some versions do not include EEPROM), and I/O pins from 16 to 70.

2.1.1 Microprocessor Unit (MPU)

A typical MPU is divided into three segments: Arithmetic Logic Unit (ALU), Registers, and Control Unit, as shown in Figure 1-3 in Chapter 1. Figure 2-2 shows a further expansion of that block diagram, specific to the microprocessor unit (MPU) in the PIC controllers. It also shows various internal busses that connect the MPU to memory.

Arithmetic Logic Unit (ALU)

The ALU includes electronic circuits (such as the adder, comparator, and flags) that are designed to perform arithmetic and logic functions, such as addition, subtraction, logical AND, OR, and Exclusive OR. The register that is used to perform these functions is generally called the accumulator. In this microcontroller family, it is called the Working Register (WREG), and it is 8 bits wide. The ALU also shows the block called Instruction Decoder, which is used to interpret an instruction; this is an internal function to the ALU. When the MPU fetches an instruction from memory, it places the instruction in the Instruction Decoder, which decodes the instruction and directs the processor on how to execute that instruction. This process is similar

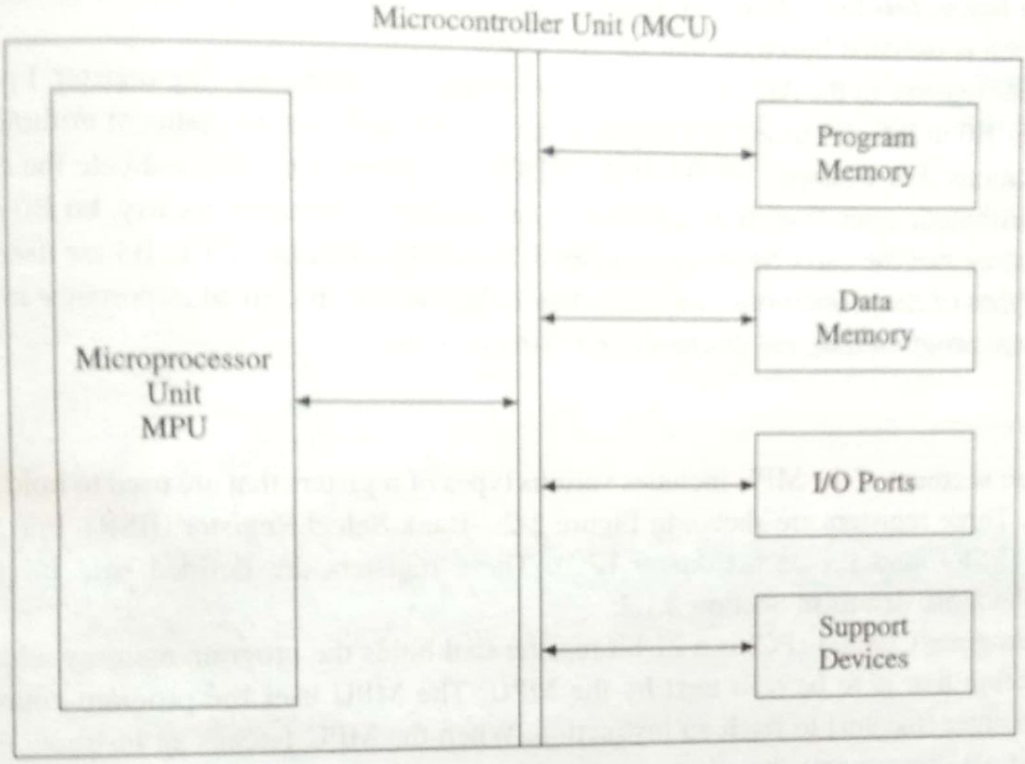


FIGURE 2-1 Block Diagram—Microcontroller Unit (MCU)

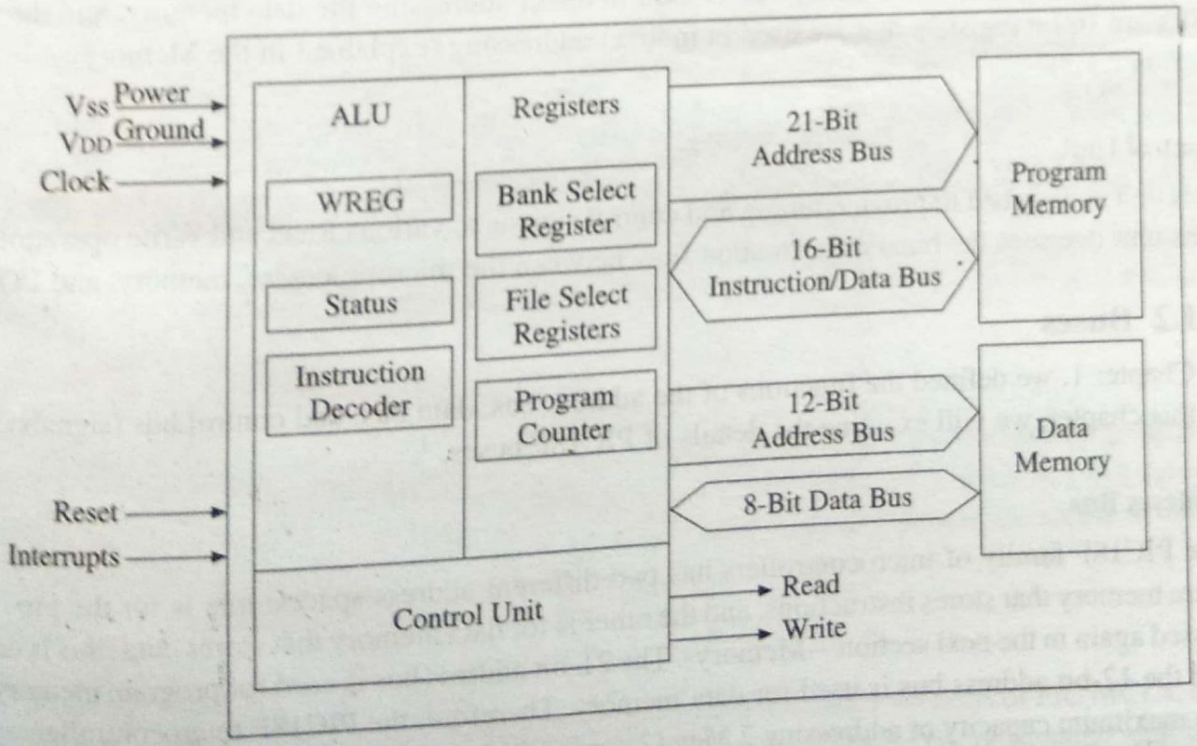


FIGURE 2-2 Block Diagram—MPU and Memory

to how our brains function. When we hear a word or a sentence, it is interpreted by the brain, and an action is initiated based on that interpretation.

Another register in the ALU is a STATUS register, also called the flag register. Five bits of this register B0 to B4 (discussed in Chapter 3) are used to indicate the status of arithmetic and logic operations. For example, bit B0 of the STATUS register is used to indicate the carry status of an arithmetic operation. If an addition of two numbers generates a carry, bit B0 is set to one, indicating that the carry has been generated. Similarly, other bits B1 to B4 are used as flags for other types of data conditions or results. These flags and their critical importance in assembly language programming are discussed in Chapters 3 and 5.

Registers

The register segment of the MPU includes various types of registers that are used to hold memory addresses. Three registers are shown in Figure 2-2—Bank Select Register (BSR), File Select Registers (FSR), and Program Counter (PC). These registers are defined here briefly, but explained in more details in Section 2.1.2.

The Program Counter (PC) is a 21-bit register that holds the program memory address of the instruction that is to be read next by the MPU. The MPU uses the program counter as a memory pointer (locator) to fetch an instruction. When the MPU fetches an instruction from memory, it also increments the address in the Program Counter to point to the next memory location.

The BSR is a four-bit register that is used in direct addressing the data memory, and the FSRs are 16-bit registers that are used in indirect addressing (explained in the Memory Section 2.1.3).

Control Unit

This unit is designed to provide timing and control signals to various Read and Write operations. This unit oversees the binary information flow between the microprocessor, memory, and I/O.

2.1.2 Buses

In Chapter 1, we defined the functions of the address bus, data bus, and control bus (signals). In this chapter, we will examine the details of PIC18F buses.

Address Bus

The PIC18F family of microcontrollers has two different address spaces: one is for the program memory that stores instructions, and the other is for data memory that stores data (this is discussed again in the next section—Memory). The 21-bit address bus is used for program memory and the 12-bit address bus is used for data memory. Therefore, the PIC18F microcontroller has maximum capacity of addressing 2 Meg ($2^{21} = 2097152$) memory registers for programs and 4 K ($2^{12} = 4096$) memory registers for data.

CHAPTER 9

INTRODUCTION TO 8051 MICROCONTROLLERS

LEARNING OUTCOMES

After studying this chapter, you will be able to understand the following:

- Differences between microprocessors and microcontrollers
- Description and features of Intel MCS-51 series microcontrollers
- Intel 8051 microcontroller architecture and features
- Power control modes of the 8051
- Stack operation in the 8051

9.1 INTRODUCTION

The microprocessor is a programmable chip that forms the CPU of a computer. Nowadays, many microprocessor chips are available in the market for users to select from depending on the application. In general, processor chips can be classified as general-purpose microprocessors, microcontrollers, and DSP processors.

A general-purpose microprocessor is the CPU of a digital computer and needs external components such as memory, input devices, output devices, and decoders to function as a microcomputer system. These chips can be used to suit any general-purpose application and can be configured by the user. Examples of 8-bit processors are Intel's 8085, Zilog 80, and Motorola 6800. Examples of 16-bit processors are Intel's 8086 and 8088 and Motorola's 68000 and examples of 32-bit processors are Intel's 80186, 80286, and 80386, and Motorola's 68030. In general, these microprocessor-based systems get data from mass storage devices, perform calculations, and store the results in storage devices. General-purpose microprocessors use external memory and a lot of processor time is involved in data transfer between the external memory and the processor.

Microcontrollers are processor chips that generally have memory, input ports, and output ports within the chip itself. Therefore, they can also be called single-chip computers, computer-on-a-chip, or system-on-a-chip. Microcontrollers are used in machine control applications, where there is no need to change the program. Equipments that use microcontrollers include computer printers, plotters, fax machines, Xerox machines, telephones, automotive engine control mechanisms, and electronic instruments such as oscilloscopes, multimeters, planimeters, IC testers, etc. The major difference between microprocessors and microcontrollers is that microcontrollers are comparatively faster because of reduced external memory accessing. Intel's 8031, 8051, and 8096 and Motorola's 68HC11 are examples of microcontrollers.

DSP processors are processing chips that have flexibility in hardware and software, to implement signal-processing algorithms. These processors have an extended arithmetic and logic unit for operation on floating or fixed point number formats. Like microcontrollers, DSP processors also have on-chip memory, I/O ports, A/D converters, and serial ports. They are used in mobile phones, digital cameras, PBX systems, and smart card readers.

Chapters 9–12 give a complete idea about the Intel 8-bit microcontroller's architecture, instruction set, programming, and hardware interfacing. Readers of these chapters are expected to know the fundamentals of microprocessors, memory structures, and assembly language programming.

9.2 INTEL'S MCS-51 SERIES MICROCONTROLLERS

Intel Corporation has many microcontrollers in both 8-bit and 16-bit configurations. The 8-bit microcontrollers are available in many part numbers, with MCS-51 as the family name. Figure 9.1 shows the various microcontrollers in the MCS-51 series, with their constituent differences. For example, 8XC51RD comes with an internal ROM of 64 KB, while 8XC51FC comes with only a 32 KB ROM.

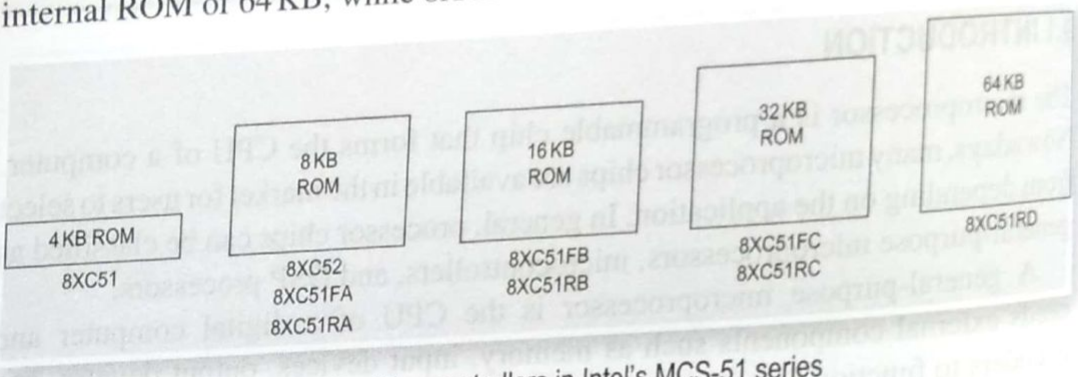


Fig. 9.1 Microcontrollers in Intel's MCS-51 series

Table 9.1 lists the features of the 8051 family of microcontrollers.

Table 9.1 Features of the 8051 family of microcontrollers

Device number	Data bus width (bits)	RAM capacity (bytes)	ROM capacity
8031	8	128	Nil
8051	8	128	4 KB
8751H	8	128	4 KB EPROM
8052AH	8	256	8 KB
8752BH	8	256	8 KB EPROM

All the microcontroller chips listed in Table 9.1 have the same basic architecture. The Intel 8051 is considered for further discussion in the topics that deal with instruction set, and software and hardware interfacing in Chapters 9–12.

9.3 INTEL 8051 ARCHITECTURE

The main features available in the 8051 chips are as follows:

(i) 8-bit CPU

(ii) 4 KB of on-chip program memory

(iii) 128 bytes of on-chip data RAM (iv) four ports of eight bits each
(v) two 16-bit timers (vi) full-duplex serial port
(vii) on-chip clock oscillator

Figures 9.2 (a) and 9.2 (b) show the architecture and block diagram of the 8051, respectively.

In addition to these features, the 8051 provides Boolean processing, six interrupt capabilities, and an 8-bit CPU for control applications.

The 8051 is an 8-bit microcontroller, i.e., the data bus within and outside the chip is eight bits wide. The address bus of the 8051 is 16 bits wide. So it can address 64 KB of memory. The lower-order address bus is multiplexed with the data bus, as in the 8085 processor. The port 0 and port 2 pins of the 8051 form the multiplexed address and data bus.

The 8051 is a 40-pin chip. The power supply $+V_{CC}$ and V_{SS} takes two pins and the built-in clock oscillator requires two pins (–XTAL1 and XTAL2) for connecting the crystal.

The four control signal pins of the 8051 are PSEN, ALE, EA, and RST as shown in Fig. 9.3 on page 307. RST is an active-high reset signal used to restart the controller chip. The 8051 responds to an RST high input only if the RST is held high for at least two machine cycles. A machine cycle is the period taken by any processor to fetch and execute one instruction. In the 8051, the maximum number of clock cycles taken for a machine cycle is 12. So the RST pin must be high for at least

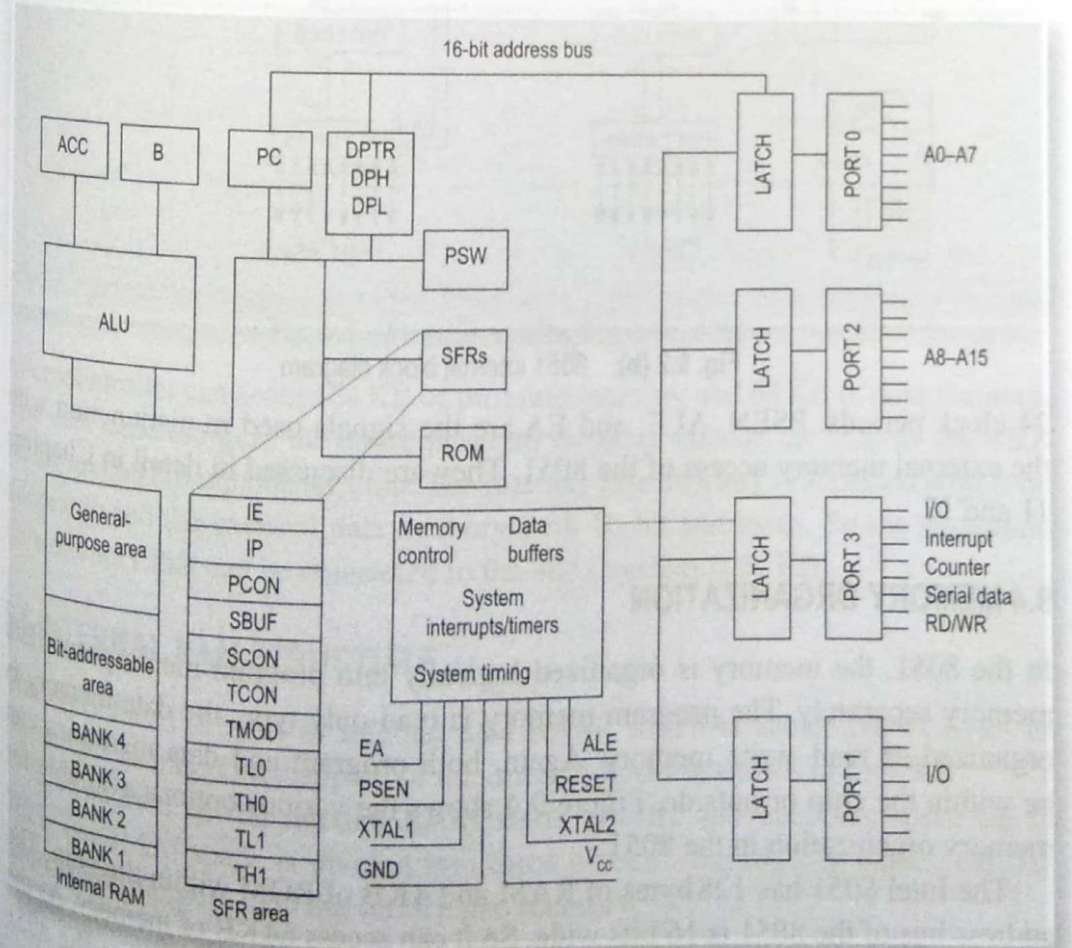


Fig. 9.2 (a) 8051 architecture

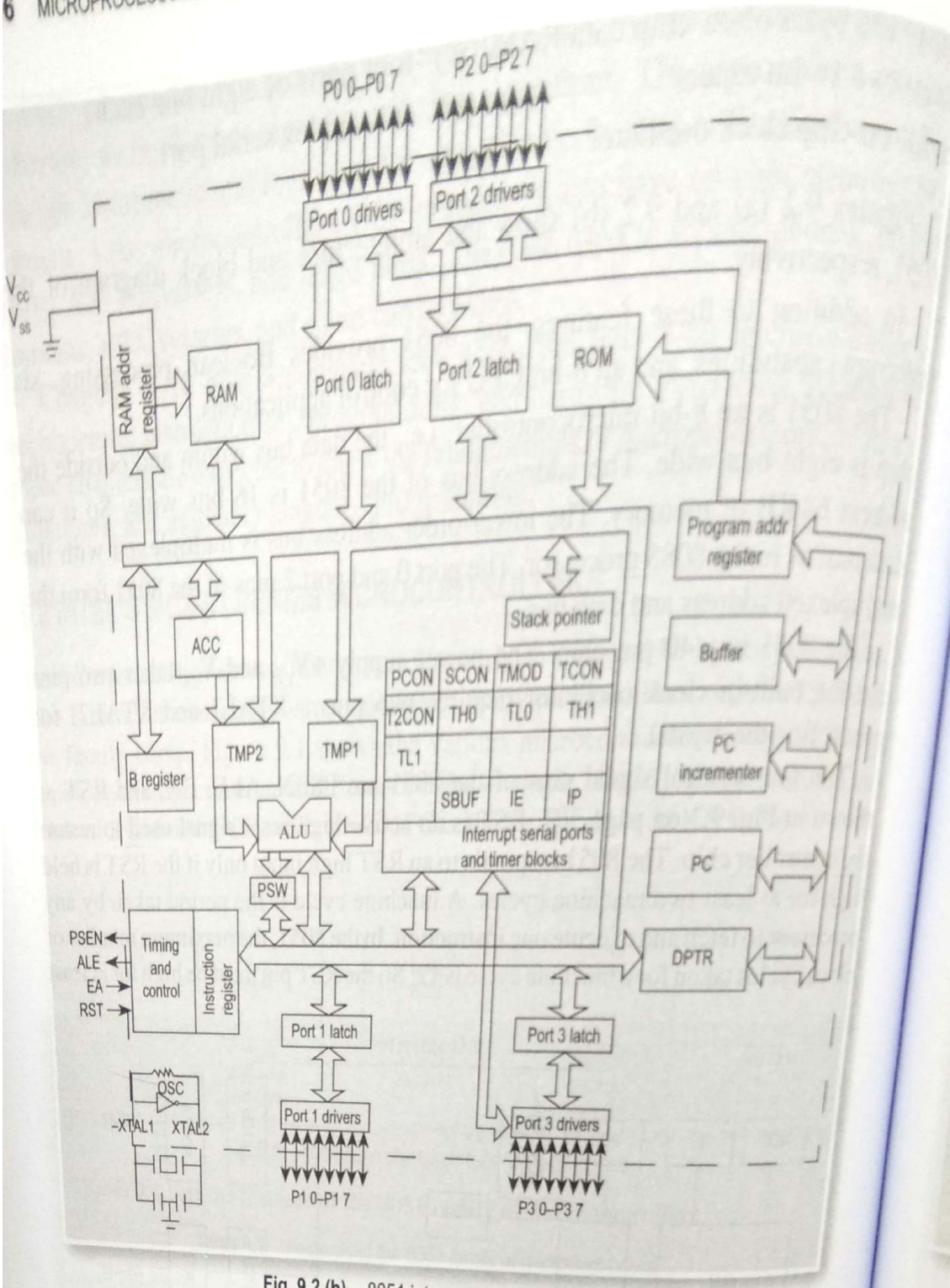


Fig. 9.2 (b) 8051 internal block diagram

24 clock periods. PSEN, ALE, and EA are the signals used in conjunction with the external memory access of the 8051. They are discussed in detail in Chapters 11 and 12.

9.4 MEMORY ORGANIZATION

In the 8051, the memory is organized logically into program memory and data memory separately. The program memory is read-only type; the data memory is organized as read–write memory. Again, both program and data memories can be within the chip or outside. Figure 9.4 shows the various options available for memory organization in the 8051.

The Intel 8051 has 128 bytes of RAM and 4 KB of ROM within the chip. The address bus of the 8051 is 16 bits wide. So it can access 64 KB of memory. As the memory is organized separately as program memory and data memory, the 8051

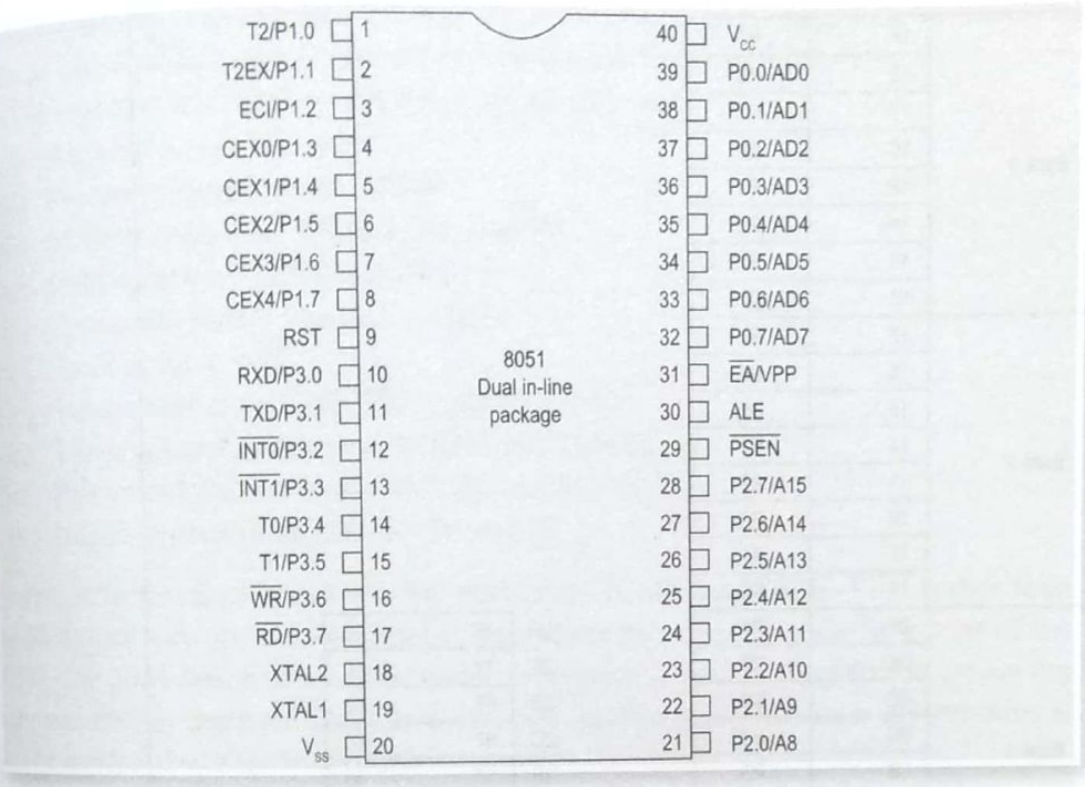


Fig. 9.3 Pin details of 8051 DIP IC

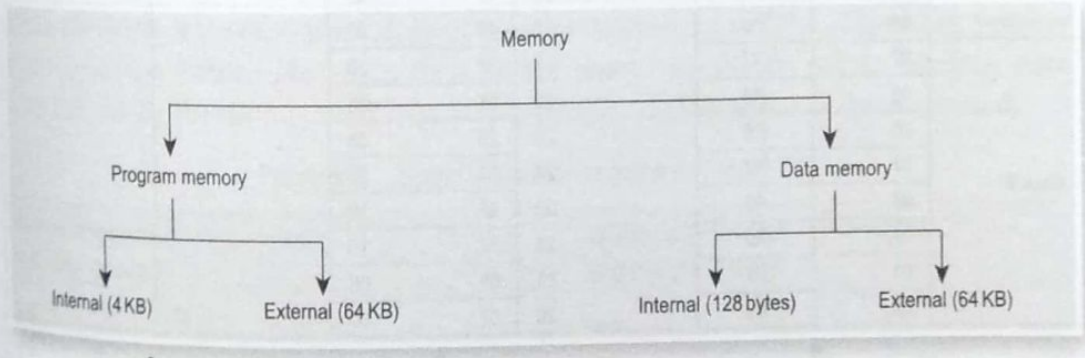


Fig. 9.4 Memory organization in the 8051

microcontroller can access 64 KB of program memory and 64 KB of data memory. The user can configure the entire program memory outside the chip or use 4 KB inside and 60 KB outside the chip. The internal data memory is accessed with 8-bit addresses and the external data memory with 16-bit addresses. So the maximum data memory that can be connected to the 8051 system is 64 KB.

9.5 INTERNAL RAM STRUCTURE

The 8051 has 128 bytes of internal data RAM, which is accessible as bytes or sometimes as bits. The mapping of the internal RAM is shown in Fig. 9.5.

The address of the internal RAM starts at 00H and occupies space up to 7FH. The RAM space is divided into three blocks—the register banks, the bit-addressable memory, and the scratch pad memory.

The 8051 has four register banks of eight registers each, with addresses from 00H to 1FH. In assembly language, they are addressed by the names R0–R7. The

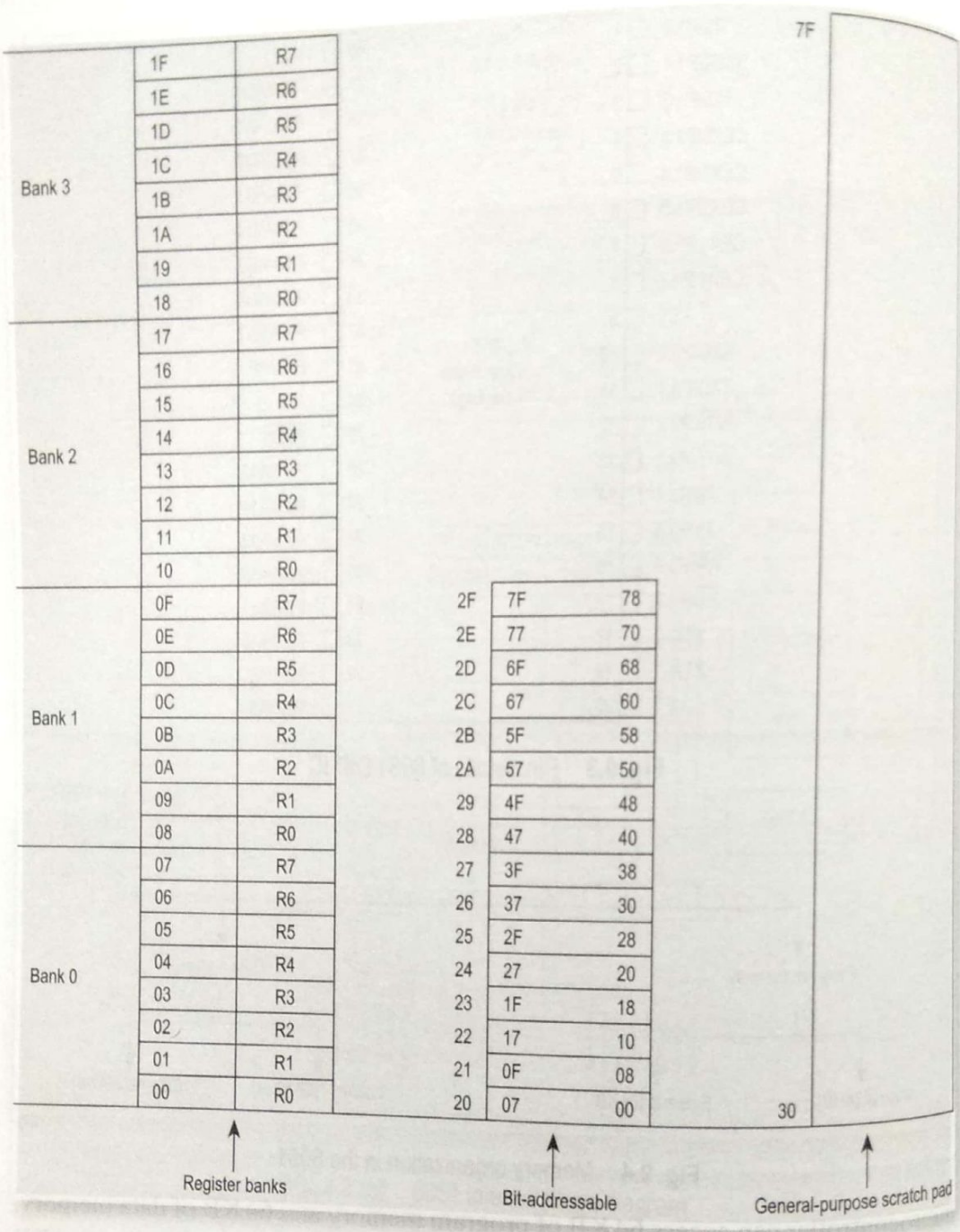


Fig. 9.5 8051 internal RAM map

register banks are identified with two bits in the processor status word (PSW). The PSW has two bits for identifying the register bank, i.e., 00 represents bank 0, 01 represents bank 1, 10 represents bank 2, and 11 represents bank 3.

In the 8051, bitwise operations are also possible with special instructions using the bit addresses. The bit-addressable memory is both bit-addressable (from 00H to 7FH) and byte-addressable (from 20H to 2FH). Bit operations are helpful in many control algorithms.

Using general-purpose scratch pad memory, programmers can read and write data at any time for any purpose. This memory ranges from the byte address 30H to the address 7FH.

9.5.1 Special Function Registers

Special function registers (SFRs), which occupy the upper 128 bytes of the internal

memory, are the registers that control the entire processor. They can be accessed only by direct addressing. The common SFRs are listed in Table 9.2.

The registers available in the 8051 are as follows:

(i) Accumulators—A and B
(ii) Processor status word—PSW
(iii) I/O port registers—P0, P1, P2, and P3
(iv) Data pointers—DPH and DPL
(v) Serial data buffer register—SBUF
(vi) Stack pointer—SP
(vii) Timer registers—TH0, TH1 and TL0, TL1
(viii) Timer control registers—TCON and TMOD
(ix) Power and port control—PCON and SCON
(x) Interrupt control registers—IP and IE

Programmers should not use the addresses in the range 80H–FFH (other than the SFRs), as they are used by Intel Corporation for expanding the functions of the 8051. The 8051 has two accumulators—registers A and B. Register B forms the accumulator for multiplication and division instructions; for other instructions it can be accessed as a general-purpose register.

The stack in the 8051 is organized within the internal RAM area. The stack pointer is eight bits wide and has to be initialized with an address in the RAM area. When the 8051 is reset, the stack pointer is by default set to 07H. The stack pointer is incremented before storing a data in the stack. Similarly, while reading data from the stack, the data is read first and then the stack pointer is decremented.

Table 9.2 Special function registers of 8051

Direct addressed memory address	SFR	Direct addressed memory address	SFR
80	P0	90	P1
81	SP	98	SCON
82	DPL	99	SBUF
83	DPH	A0	P2
88	TCON	A8	IE
89	TMOD	B0	P3
8A	TL0	B8	IP
8B	TL1	D0	PSW
8C	TH0	E0	ACC
8D	TH1	F0	B

9.5.2 Processor Status Word

The PSW contains all the flags of the 8051 and is eight bits wide. The bit pattern of this flag register is given in Table 9.3. From the table, it can be seen that there are seven flag bits in the PSW of the 8051. The PSW is accessible fully as an 8-bit register, with the address D0H. Meanwhile, individual bits of the PSW can be

accessed with the bit addresses given in Table 9.3. The contents of the PSW upon reset are given in the third row.

Table 9.3 PSW format of 8051

PSW	CY	AC	F0	RS1	RS0	OV	–	P
Bit address	D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H
Contents upon reset	0	0	0	0	0	0	X	0

Parity bit (P) It is set to 1 if the accumulator contains an odd number of 1s, after an arithmetic or logical operation.

Overflow flag (OV) This flag is set during ALU operations, to indicate overflow in the result. It is set to 1 if there is a carry out of either the D7 bit or the D6 bit of the accumulator. Overflow flag is set when arithmetic operations such as add and subtract result in sign conflict.

The conditions under which the OV flag is set are as follows:

Positive + Positive = Negative
Negative + Negative = Positive
Positive – Negative = Negative
Negative – Positive = Positive

Register bank select bits (RS0 and RS1) These bits are user-programmable. They can be set by the programmer to point to the correct register banks. The register bank selection in the programs can be changed using these two bits.

Table 9.4 explains how these bits can be changed to select the appropriate register bank.

Table 9.4 Register bank selection using PSW

RS1	RS0	Selected bank	Address range
0	0	Bank 0	00H–07H
0	1	Bank 1	08H–0FH
1	0	Bank 2	10H–17H
1	1	Bank 3	18H–1FH

General-purpose flag (F0) This is a user-programmable flag; the user can program and store any bit of his/her choice in this flag, using the bit address.

Auxiliary carry flag (AC) It is used in association with BCD arithmetic. This flag is set when there is a carry out of the D3 bit of the accumulator.

Carry flag (CY) This flag is used to indicate the carry generated after arithmetic operations. It can also be used as an accumulator, to store one of the data bits for bit-related Boolean instructions.

The 8051 supports bit manipulation instructions. This means that in addition to the byte operations, bit operations can also be done using bit data. For this purpose, the contents of the PSW are bit-addressable. Similarly, the contents of

the accumulator and register B are also bit-addressable. The bit addresses of all the bits of the accumulator and register B are given in Tables 9.5 (a) and 9.5 (b).

Table 9.5 (a) Addresses and contents of accumulator bits

Accumulator bits	ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0
Bit address	E7	E6	E5	E4	E3	E2	E1	E0
Contents upon reset	0	0	0	0	0	0	0	0

Table 9.5 (b) Addresses and contents of register B bits

Register B bits	B.7	B.6	B.5	B.4	B.3	B.2	B.1	B.0
Bit address	F7	F6	F5	F4	F3	F2	F1	F0
Contents upon reset	0	0	0	0	0	0	0	0

9.6 POWER CONTROL IN 8051

The 8051 has various power control modes, which are used to control the power consumed by the microcontroller chip. Some of these modes let the microcontroller go into a 'sleep' mode, which makes it consume lesser power than during normal operation. The power control modes are selected through the Special Function Register PCON. Table 9.6 gives the bit pattern of the PCON register (87H).

Table 9.6 Bit pattern for PCON (87H) register

Bit	Name	Function
7	SMOD	Serial port baud rate set bit
6	–	Reserved
5	–	Reserved
4	–	Reserved
3	GF1	General-purpose flag 1
2	GF0	General-purpose flag 0
1	PD	Power down mode set bit
0	IPL	Idle mode set bit

9.6.1 Idle Mode

The microcontroller enters the idle mode whenever the PCON.0 bit is set to 1. In the idle mode, the clock pulses applied to the CPU are masked, while all other units such as interrupt controllers, etc. are kept active. The contents of the CPU are not affected in the idle mode. The processor can be revoked from the idle mode by applying either a hardware interrupt or a hardware reset signal. These two

actions will reset PCON.0 and processor execution will resume at the instruction following the instruction that set the idle mode.

9.6.2 Power Down Mode

The power down mode is initiated by making PCON.1 bit 1. In this mode, the clock generator is switched off and only the internal memory is active. The supply voltage V_{cc} can be reduced to 2 V and the power consumption can be reduced. The only way to revoke the processor from power down mode is to reset the system.

9.7 STACK OPERATION

In the 8051, the stack is configured as a series of memory locations following the Last-In First-Out (LIFO) pattern. In general, the stack is initialized in the internal RAM area. Any 8-bit data can be stored and retrieved from the stack using PUSH and POP instructions, with the help of the stack pointer.

The stack pointer (SP) is an 8-bit register within the SFR area, with the address 81H. This register can hold one 8-bit address at a time, which is actually the memory location at top of the stack. A push operation in the 8051 is used to store an 8-bit data in the stack. The PUSH instruction first increments the value of SP and then stores the data mentioned in the instruction in the memory location pointed to by SP. Similarly, the POP instruction stores the value from the top of the stack in the register mentioned in the instruction and then decrements the value of SP. The stack pointer is initialized to the value 07H when the 8051 microcontroller is reset. The other instructions of the 8051 that affect the stack and the stack pointer are ACALL, LCALL, RET, and RETI. The stack pointer can be initialized to any internal RAM address by the programmer, by writing the required address in the SP SFR address 81H.

POINTS TO REMEMBER

- Intel MCS-51 is a family of microcontrollers and 8051 is the familiar IC in the family. Different versions of the basic 8051 are available from Intel and other chip manufacturers.
- The 8051 is an 8-bit microcontroller with built-in RAM, I/O ports, timers, and interrupt facility.
- The functioning of the microcontroller is controlled using a set of registers called special function registers (SFRs).
- The 8051 can itself act as an independent system, with all memory and ports inside it. If necessary, memory and I/O ports can be added externally.
- The processor supports many operating modes such as power down mode, idle mode, etc.

CHAPTER 1
ARM EMBEDDED SYSTEMS

The ARM processor core is a key component of many successful 32-bit embedded systems. You probably own one yourself and may not even realize it! ARM cores are widely used in mobile phones, handheld organizers, and a multitude of other everyday portable consumer devices.

ARM's designers have come a long way from the first ARM1 prototype in 1985. Over one billion ARM processors had been shipped worldwide by the end of 2001. The ARM company bases their success on a simple and powerful original design, which continues to improve today through constant technical innovation. In fact, the ARM core is not a single core, but a whole family of designs sharing similar design principles and a common instruction set.

For example, one of ARM's most successful cores is the ARM7TDMI. It provides up to 120 Dhrystone MIPS[1] and is known for its high code density and low power consumption, making it ideal for mobile embedded devices.

In this first chapter we discuss how the RISC (reduced instruction set computer) design philosophy was adapted by ARM to create a flexible embedded processor. We then introduce an example embedded device and discuss the typical hardware and software technologies that surround an ARM processor.

---

1. Dhrystone MIPS version 2.1 is a small benchmarking program.

1.1 THE RISC DESIGN PHILOSOPHY

The ARM core uses a RISC architecture. RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed. The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware. As a result, a RISC design places greater demands on the compiler. In contrast, the traditional complex instruction set computer (CISC) relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated. Figure 1.1 illustrates these major differences.

The RISC philosophy is implemented with four major design rules:

1. *Instructions*—RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction. In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.
2. *Pipelines*—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage. There is no need for an instruction to be executed by a miniprogram called microcode as on CISC processors.
3. *Registers*—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data

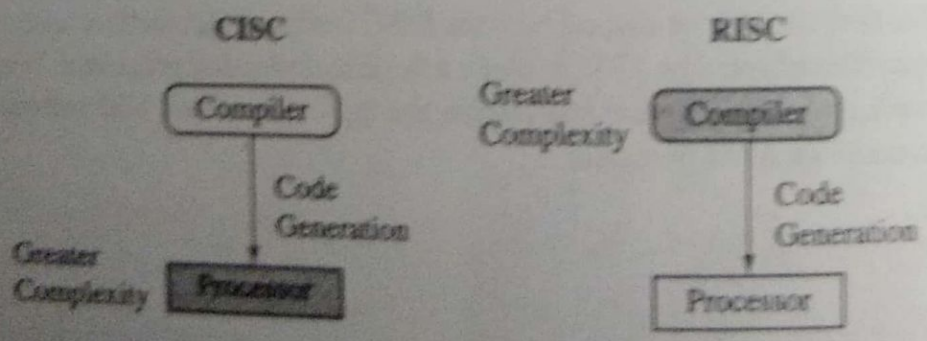


Figure 1.1 CISC vs. RISC. CISC emphasizes hardware complexity. RISC emphasizes compiler complexity.

processing operations. In contrast, CISC processors have dedicated registers for specific purposes.

4. *Load-store architecture*—The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses. In contrast, with a CISC design the data processing operations can act on memory directly.

These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies. In contrast, traditional CISC processors are more complex and operate at lower clock frequencies. Over the course of two decades, however, the distinction between RISC and CISC has blurred as CISC processors have implemented more RISC concepts.

1.2 The ARM Design Philosophy

There are a number of physical features that have driven the ARM processor design. First, portable embedded systems require some form of battery power. The ARM processor has been specifically designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).

High code density is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions. High code density is useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.

In addition, embedded systems are price sensitive and use slow and low-cost memory devices. For high-volume applications like digital cameras, every cent has to be accounted for in the design. The ability to use low-cost memory devices produces substantial savings.

Another important requirement is to reduce the area of the die taken up by the embedded processor. For a single-chip solution, the smaller the area used by the embedded processor, the more available space for specialized peripherals. This in turn reduces the cost of the design and manufacturing since fewer discrete chips are required for the end product.

ARM has incorporated hardware debug technology within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve issues faster, which has a direct effect on the time to market and reduces overall development costs.

The ARM core is not a pure RISC architecture because of the constraints of its primary application—the embedded system. In some sense, the strength of the ARM core is that it does not take the RISC concept too far. In today's systems the key is not raw processor speed but total effective system performance and power consumption.

1.2.1 Instruction Set for Embedded Systems

The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:

- *Variable cycle execution for certain instructions*—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses, which increases performance since sequential memory accesses are often faster than random accesses. Code density is also improved since multiple register transfers are common operations at the start and end of functions.
- *Inline barrel shifter leading to more complex instructions*—The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density. We explain this feature in more detail in Chapters 2, 3, and 4.
- *Thumb 16-bit instruction set*—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions. The 16-bit instructions improve code density by about 30% over 32-bit fixed-length instructions.
- *Conditional execution*—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.
- *Enhanced instructions*—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16-bit multiplier operations and saturation. These instructions allow a faster-performing ARM processor in some cases to replace the traditional combinations of a processor plus a DSP.

These additional features have made the ARM processor one of the most commonly used 32-bit embedded processor cores. Many of the top semiconductor companies around the world produce products based around the ARM processor.

1.3 Embedded System Hardware

Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems used on a NASA space probe. All these devices use a combination of software and hardware components. Each component is chosen for efficiency and, if applicable, is designed for future extension and expansion.

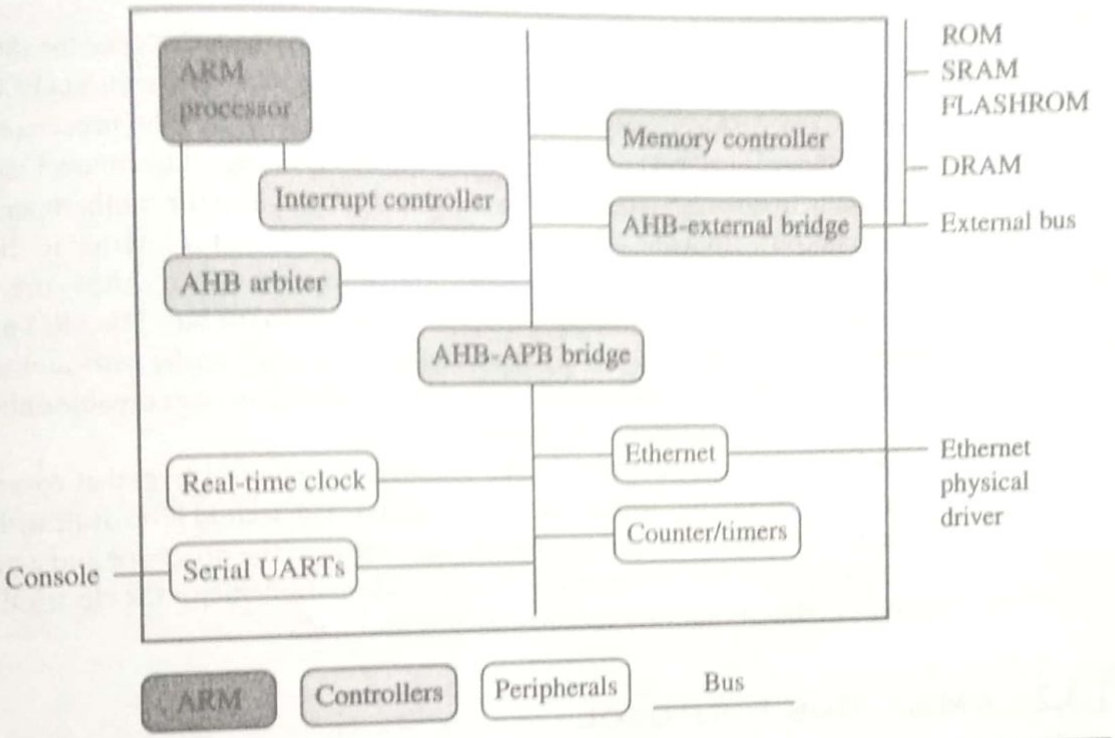


Figure 1.2 An example of an ARM-based embedded device, a microcontroller.

Figure 1.2 shows a typical embedded device based on an ARM core. Each box represents a feature or function. The lines connecting the boxes are the buses carrying data. We can separate the device into four main hardware components:

- The *ARM processor* controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components that interface it with a bus. These components can include memory management and caches.
- *Controllers* coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.
- The *peripherals* provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.
- A *bus* is used to communicate between different parts of the device.

1.3.1 ARM Bus Technology

Embedded systems use different bus technologies than those designed for x86 PCs. The most common PC bus technology, the Peripheral Component Interconnect (PCI) bus, connects such devices as video cards and hard disk controllers to the x86 processor bus. This type of technology is external or off-chip (i.e., the bus is designed to connect mechanically and electrically to devices external to the chip) and is built into the motherboard of a PC.

In contrast, embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

There are two different classes of devices attached to the bus. The ARM processor core is a *bus master*—a logical device capable of initiating a data transfer with another device across the same bus. Peripherals tend to be *bus slaves*—logical devices capable only of responding to a transfer request from a bus master device.

A bus has two architecture levels. The first is a physical level that covers the electrical characteristics and bus width (16, 32, or 64 bits). The second level deals with *protocol*—the logical rules that govern the communication between the processor and a peripheral.

ARM is primarily a design company. It seldom implements the electrical characteristics of the bus, but it routinely specifies the bus protocol.

1.3.2 AMBA Bus Protocol

The Advanced Microcontroller Bus Architecture (AMBA) was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors. The first AMBA buses introduced were the ARM System Bus (ASB) and the ARM Peripheral Bus (APB). Later ARM introduced another bus design, called the ARM High Performance Bus (AHB). Using AMBA, peripheral designers can reuse the same design on multiple projects. Because there are a large number of peripherals developed with an AMBA interface, hardware designers have a wide choice of tested and proven peripherals for use in a device. A peripheral can simply be bolted onto the on-chip bus without having to redesign an interface for each different processor architecture. This plug-and-play interface for hardware developers improves availability and time to market.

AHB provides higher data throughput than ASB because it is based on a centralized multiplexed bus scheme rather than the ASB bidirectional bus design. This change allows the AHB bus to run at higher clock speeds and to be the first ARM bus to support widths of 64 and 128 bits. ARM has introduced two variations on the AHB bus: Multi-layer AHB and AHB-Lite. In contrast to the original AHB, which allows a single bus master to be active on the bus at any time, the Multi-layer AHB bus allows multiple active bus masters. AHB-Lite is a subset of the AHB bus and it is limited to a single bus master. This bus was developed for designs that do not require the full features of the standard AHB bus.

AHB and Multi-layer AHB support the same protocol for master and slave but have different interconnects. The new interconnects in Multi-layer AHB are good for systems with multiple processors. They permit operations to occur in parallel and allow for higher throughput rates.

The example device shown in Figure 1.2 has three buses: an AHB bus for the high-performance peripherals, an APB bus for the slower peripherals, and a third bus for external peripherals, proprietary to this device. This external bus requires a specialized bridge to connect with the AHB bus.

1.3.3 MEMORY

An embedded system has to have some form of memory to store and execute code. You have to compare price, performance, and power consumption when deciding upon specific memory characteristics, such as hierarchy, width, and type. If memory has to run twice as fast to maintain a desired bandwidth, then the memory power requirement may be higher.

1.3.3.1 Hierarchy

All computer systems have memory arranged in some form of hierarchy. Figure 1.2 shows a device that supports external off-chip memory. Internal to the processor there is an option of a cache (not shown in Figure 1.2) to improve memory performance.

Figure 1.3 shows the memory trade-offs: the fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away. Generally the closer memory is to the processor core, the more it costs and the smaller its capacity.

The cache is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. A cache provides an overall increase in performance but with a loss of predictable execution time. Although the cache increases the

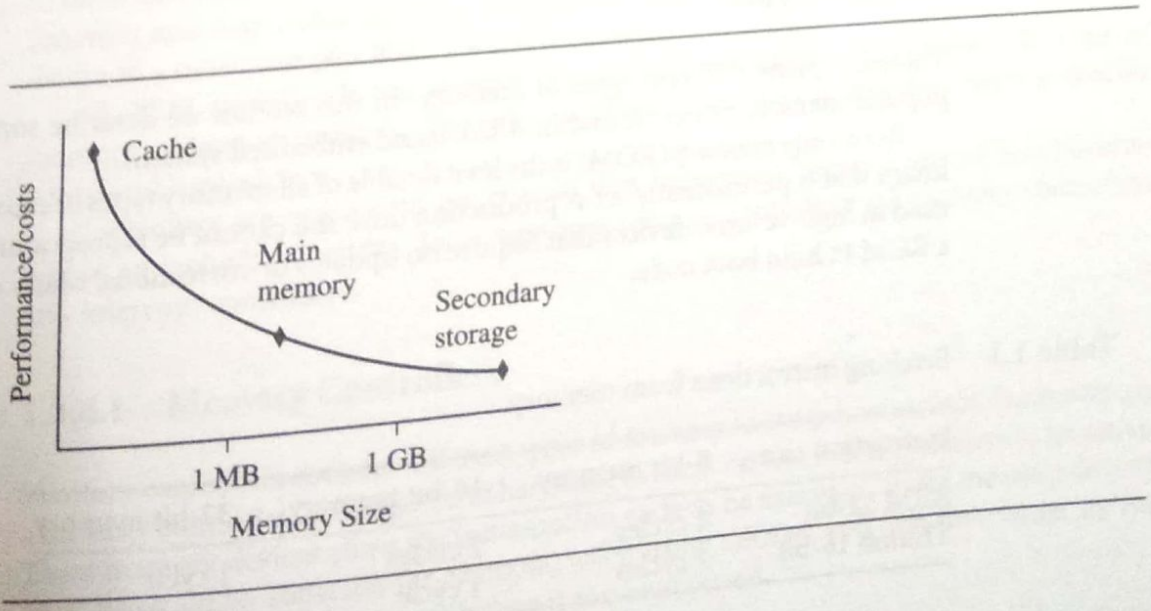


Figure 1.3 Storage trade-offs.

general performance of the system, it does not help real-time system response. Note that many small embedded systems do not require the performance benefits of a cache.

The main memory is large—around 256 KB to 256 MB (or even greater), depending on the application—and is generally stored in separate chips. Load and store instructions access the main memory unless the values have been stored in the cache for fast access. Secondary storage is the largest and slowest form of memory. Hard disk drives and CD-ROM drives are examples of secondary storage. These days secondary storage may vary from 600 MB to 60 GB.

1.3.3.2 Width

The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits. The memory width has a direct effect on the overall performance and cost ratio.

If you have an uncached system using 32-bit ARM instructions and 16-bit-wide memory chips, then the processor will have to make two memory fetches per instruction. Each fetch requires two 16-bit loads. This obviously has the effect of reducing system performance, but the benefit is that 16-bit memory is less expensive.

In contrast, if the core executes 16-bit Thumb instructions, it will achieve better performance with a 16-bit memory. The higher performance is a result of the core making only a single fetch to memory to load an instruction. Hence, using Thumb instructions with 16-bit-wide memory devices provides both improved performance and reduced cost.

Table 1.1 summarizes theoretical cycle times on an ARM processor using different memory width devices.

1.3.3.3 Types

There are many different types of memory. In this section we describe some of the more popular memory devices found in ARM-based embedded systems.

Read-only memory (ROM) is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed. ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code.

Table 1.1 Fetching instructions from memory.

Instruction size	8-bit memory	16-bit memory	32-bit memory
ARM 32-bit	4 cycles	2 cycles	1 cycle
Thumb 16-bit	2 cycles	1 cycle	1 cycle

Flash ROM can be written to as well as read, but it is slow to write so you shouldn't use it for holding dynamic data. Its main use is for holding the device firmware or storing long-term data that needs to be preserved after power is off. The erasing and writing of flash ROM are completely software controlled with no additional hardware circuity required, which reduces the manufacturing costs. Flash ROM has become the most popular of the read-only memory types and is currently being used as an alternative for mass or secondary storage.

Dynamic random access memory (DRAM) is the most commonly used RAM for devices. It has the lowest cost per megabyte compared with other types of RAM. DRAM is *dynamic*—it needs to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controller before using the memory.

Static random access memory (SRAM) is faster than the more traditional DRAM, but requires more silicon area. SRAM is *static*—the RAM does not require refreshing. The access time for SRAM is considerably shorter than the equivalent DRAM because SRAM does not require a pause between data accesses. Because of its higher cost, it is used mostly for smaller high-speed tasks, such as fast memory and caches.

Synchronous dynamic random access memory (SDRAM) is one of many subcategories of DRAM. It can run at much higher clock speeds than conventional memory. SDRAM synchronizes itself with the processor bus because it is clocked. Internally the data is fetched from memory cells, pipelined, and finally brought out on the bus in a burst. The old-style DRAM is asynchronous, so does not burst as efficiently as SDRAM.

1.3.4 PERIPHERALS

Embedded systems that interact with the outside world need some form of peripheral device. A peripheral device performs input and output functions for the chip by connecting to other devices or sensors that are off-chip. Each peripheral device usually performs a single function and may reside on-chip. Peripherals range from a simple serial communication device to a more complex 802.11 wireless device.

All ARM peripherals are *memory mapped*—the programming interface is a set of memory-addressed registers. The address of these registers is an offset from a specific peripheral base address.

Controllers are specialized peripherals that implement higher levels of functionality within an embedded system. Two important types of controllers are memory controllers and interrupt controllers.

1.3.4.1 Memory Controllers

Memory controllers connect different types of memory to the processor bus. On power-up a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed. Some memory devices must be set up by software; for example, when using DRAM, you first have to set up the memory timings and refresh rate before it can be accessed.

1.3.4.2 Interrupt Controllers

When a peripheral or device requires attention, it raises an interrupt to the processor. An interrupt controller provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.

There are two types of interrupt controller available for the ARM processor: the standard interrupt controller and the vector interrupt controller (VIC).

The standard interrupt controller sends an interrupt signal to the processor core when an external device requests servicing. It can be programmed to ignore or mask an individual device or set of devices. The interrupt handler determines which device requires servicing by reading a device bitmap register in the interrupt controller.

The VIC is more powerful than the standard interrupt controller because it prioritizes interrupts and simplifies the determination of which device caused the interrupt. After associating a priority and a handler address with each interrupt, the VIC only asserts an interrupt signal to the core if the priority of a new interrupt is higher than the currently executing interrupt handler. Depending on its type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler for the device from the VIC, or cause the core to jump to the handler for the device directly.

1.4 Embedded System Software

An embedded system needs software to drive it. Figure 1.4 shows four typical software components required to control an embedded device. Each software component in the stack uses a higher level of abstraction to separate the code from the hardware device.

The initialization code is the first code executed on the board and is specific to a particular target or group of targets. It sets up the minimum parts of the board before handing control over to the operating system.

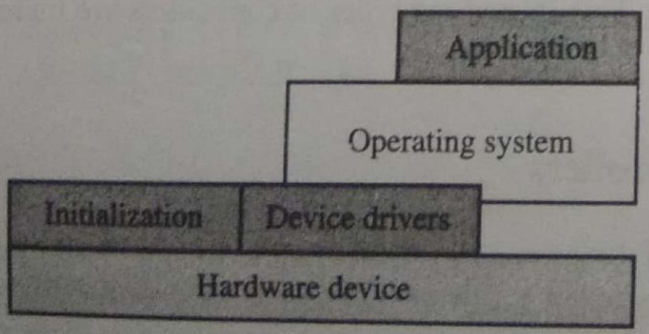


Figure 1.4 Software abstraction layers executing on hardware.

The operating system provides an infrastructure to control applications and manage hardware system resources. Many embedded systems do not require a full operating system but merely a simple task scheduler that is either event or poll driven.

The device drivers are the third component shown in Figure 1.4. They provide a consistent software interface to the peripherals on the hardware device.

Finally, an application performs one of the tasks required for a device. For example, a mobile phone might have a diary application. There may be multiple applications running on the same device, controlled by the operating system.

The software components can run from ROM or RAM. ROM code that is fixed on the device (for example, the initialization code) is called *firmware*.

1.4.1 INITIALIZATION (BOOT) CODE

Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run. It usually configures the memory controller and processor caches and initializes some devices. In a simple system the operating system might be replaced by a simple scheduler or debug monitor.

The initialization code handles a number of administrative tasks prior to handing control over to an operating system image. We can group these different tasks into three phases: initial hardware configuration, diagnostics, and booting.

Initial hardware configuration involves setting up the target platform so it can boot an image. Although the target platform itself comes up in a standard configuration, this configuration normally requires modification to satisfy the requirements of the booted image. For example, the memory system normally requires reorganization of the memory map, as shown in Example 1.1.

Diagnostics are often embedded in the initialization code. Diagnostic code tests the system by exercising the hardware target to check if the target is in working order. It also tracks down standard system-related issues. This type of testing is important for manufacturing since it occurs after the software product is complete. The primary purpose of diagnostic code is fault identification and isolation.

Booting involves loading an image and handing control over to that image. The boot process itself can be complicated if the system must boot different operating systems or different versions of the same operating system.

Booting an image is the final phase, but first you must load the image. Loading an image involves anything from copying an entire program including code and data into RAM, to just copying a data area containing volatile variables into RAM. Once booted, the system hands over control by modifying the program counter to point into the start of the image.

Sometimes, to reduce the image size, an image is compressed. The image is then decompressed either when it is loaded or when control is handed over to it.

EXAMPLE 1.1 Initializing or organizing memory is an important part of the initialization code because many operating systems expect a known memory layout before they can start.

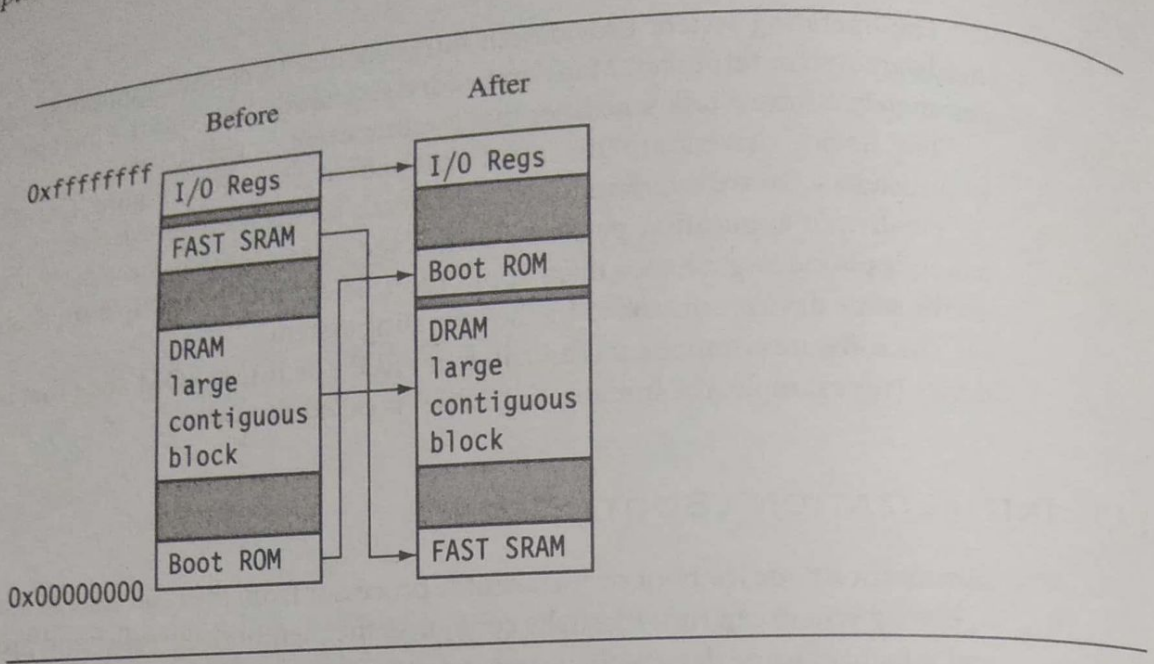


Figure 1.5 Memory remapping.

Figure 1.5 shows memory before and after reorganization. It is common for ARM-based embedded systems to provide for memory remapping because it allows the system to start the initialization code from ROM at power-up. The initialization code then redefines or remaps the memory map to place RAM at address 0x00000000—an important step because then the exception vector table can be in RAM and thus can be reprogrammed. We will discuss the vector table in more detail in Section 2.4.

1.4.2 Operating System

The initialization process prepares the hardware for an operating system to take control. An operating system organizes the system resources: the peripherals, memory, and processing time. With an operating system controlling these resources, they can be efficiently used by different applications running within the operating system environment.

ARM processors support over 50 operating systems. We can divide operating systems into two main categories: real-time operating systems (RTOSs) and platform operating systems.

RTOSs provide guaranteed response times to events. Different operating systems have different amounts of control over the system response time. A hard real-time application requires a guaranteed response to work at all. In contrast, a soft real-time application requires a good response time, but the performance degrades more gracefully if the response time overruns. Systems running an RTOS generally do not have secondary storage.

Platform operating systems require a memory management unit to manage large, non-real-time applications and tend to have secondary storage. The Linux operating system is a typical example of a platform operating system.

These two categories of operating system are not mutually exclusive: there are operating systems that use an ARM core with a memory management unit and have real-time characteristics. ARM has developed a set of processor cores that specifically target each category.

1.4.3 Applications

The operating system schedules applications—code dedicated to handling a particular task. An application implements a processing task; the operating system controls the environment. An embedded system can have one active application or several applications running simultaneously.

ARM processors are found in numerous market segments, including networking, automotive, mobile and consumer devices, mass storage, and imaging. Within each segment ARM processors can be found in multiple applications.

For example, the ARM processor is found in networking applications like home gateways, DSL modems for high-speed Internet communication, and 802.11 wireless communication. The mobile device segment is the largest application area for ARM processors because of mobile phones. ARM processors are also found in mass storage devices such as hard drives and imaging products such as inkjet printers—applications that are cost sensitive and high volume.

In contrast, ARM processors are not found in applications that require leading-edge high performance. Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

1.5 Summary

Pure RISC is aimed at high performance, but ARM uses a modified RISC design philosophy that also targets good code density and low power consumption. An embedded system consists of a processor core surrounded by caches, memory, and peripherals. The system is controlled by operating system software that manages application tasks.

The key points in a RISC design philosophy are to improve performance by reducing the complexity of instructions, to speed up instruction processing by using a pipeline, to provide a large register set to store data near the core, and to use a load-store architecture.

The ARM design philosophy also incorporates some non-RISC ideas:

- It allows variable cycle execution on certain instructions to save power, area, and code size.
- It adds a barrel shifter to expand the capability of certain instructions.
- It uses the Thumb 16-bit instruction set to improve code density.

- It improves code density and performance by conditionally executing instructions.
- It includes enhanced instructions to perform digital signal processing type functions.

An embedded system includes the following hardware components: ARM *processors* are found embedded in chips. Programmers access *peripherals* through memory-mapped registers. There is a special type of peripheral called a *controller*, which embedded systems use to configure higher-level functions such as memory and interrupts. The AMBA on-chip *bus* is used to connect the processor and peripherals together.

An embedded system also includes the following software components: *Initialization code* configures the hardware to a known state. Once configured, *operating systems* can be loaded and executed. Operating systems provide a common programming environment for the use of hardware resources and infrastructure. *Device drivers* provide a standard interface to peripherals. An *application* performs the task-specific duties of an embedded system.

CHAPTER 2
ARM PROCESSOR FUNDAMENTALS

Chapter 1 covered embedded systems with an ARM processor. In this chapter we will focus on the actual processor itself. First, we will provide an overview of the processor core and describe how data moves between its different parts. We will describe the programmer's model from a software developer's view of the ARM processor, which will show you the functions of the processor core and how different parts interact. We will also take a look at the core extensions that form an ARM processor. Core extensions speed up and organize main memory as well as extend the instruction set. We will then cover the revisions to the ARM core architecture by describing the ARM core naming conventions used to identify them and the chronological changes to the ARM instruction set architecture. The final section introduces the architecture implementations by subdividing them into specific ARM processor core families.

A programmer can think of an ARM core as functional units connected by data buses, as shown in Figure 2.1, where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area. The figure shows not only the flow of data but also the abstract components that make up an ARM core.

Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item. Figure 2.1 shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. In contrast, Harvard implementations of the ARM use two different buses.

The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.

The ARM processor, like all RISC processors, uses a *load-store architecture.* This means it has two instruction types for transferring data in and out of the processor: load instructions copy data from memory to registers in the core, and conversely the store

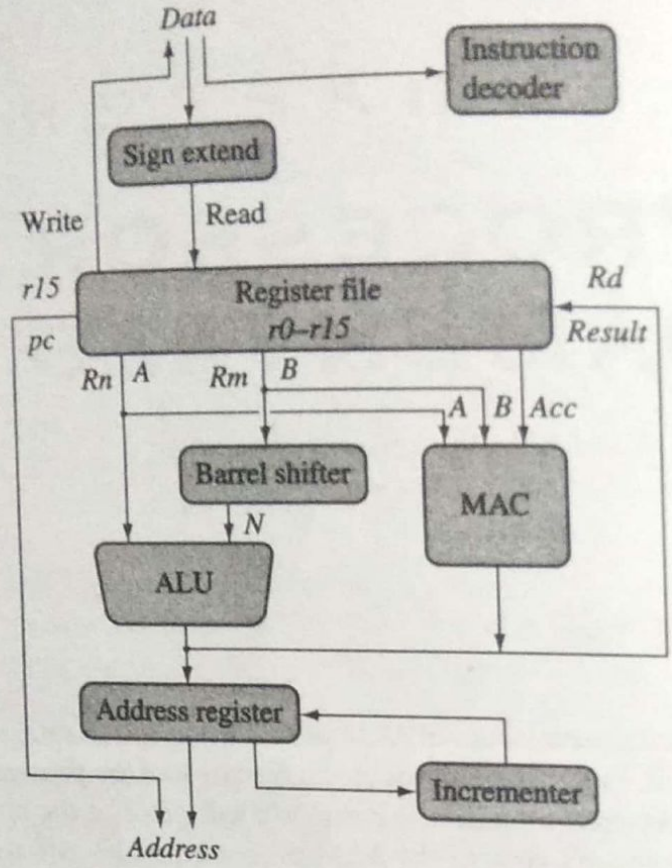


Figure 2.1 ARM core dataflow model.

instructions copy data from registers to memory. There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers.

Data items are placed in the *register file*—a storage bank made up of 32-bit registers. Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

ARM instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*. Source operands are read from the register file using the internal buses *A* and *B*, respectively.

The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result. Data processing instructions write the result in *Rd* directly to the register file. Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the *Address* bus.

One important feature of the ARM is that register *Rm* alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.

After passing through the functional units, the result in *Rd* is written back to the register file using the *Result* bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location. The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

Now that you have an overview of the processor core we'll take a more detailed look at some of the key components of the processor: the registers, the current program status register (*cpsr*), and the pipeline.

2.1 REGISTERS

General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4*. Figure 2.2 shows the active registers available in *user* mode—a protected mode normally

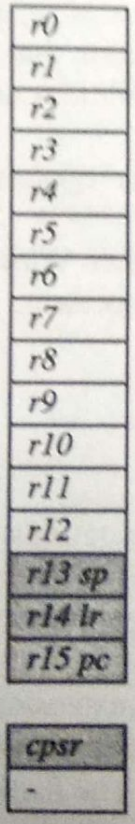


Figure 2.2 Registers available in *user* mode.

used when executing applications. The processor can operate in seven different modes, which we will introduce shortly. All the registers shown are 32 bits in size.

There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as *r0* to *r15*.

The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are frequently given different labels to differentiate them from the other registers.

In Figure 2.2, the shaded registers identify the assigned special-purpose registers:

- Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack in the current processor mode.
- Register *r14* is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.
- Register *r15* is the program counter (*pc*) and contains the address of the next instruction to be fetched by the processor.

Figure

Depending upon the context, registers *r13* and *r14* can also be used as general-purpose registers, which can be particularly useful since these registers are banked during a processor mode change. However, it is dangerous to use *r13* as a general register when the processor is running any form of operating system because operating systems often assume that *r13* always points to a valid stack frame.

2.2.1

In ARM state the registers *r0* to *r13* are *orthogonal*—any instruction that you can apply to *r0* you can equally well apply to any of the other registers. However, there are instructions that treat *r14* and *r15* in a special way.

In addition to the 16 data registers, there are two program status registers: *cpsr* and *spsr* (the current and saved program status registers, respectively).

The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

2.2 Current Program Status Register

The ARM core uses the *cpsr* to monitor and control internal operations. The *cpsr* is a dedicated 32-bit register and resides in the register file. Figure 2.3 shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

The *cpsr* is divided into four fields, each 8 bits wide: flags, status, extension, and control. In current designs the extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupt mask bits. The flags field contains the condition flags.

2.2.2

Some ARM processor cores have extra bits allocated. For example, the *J* bit, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute

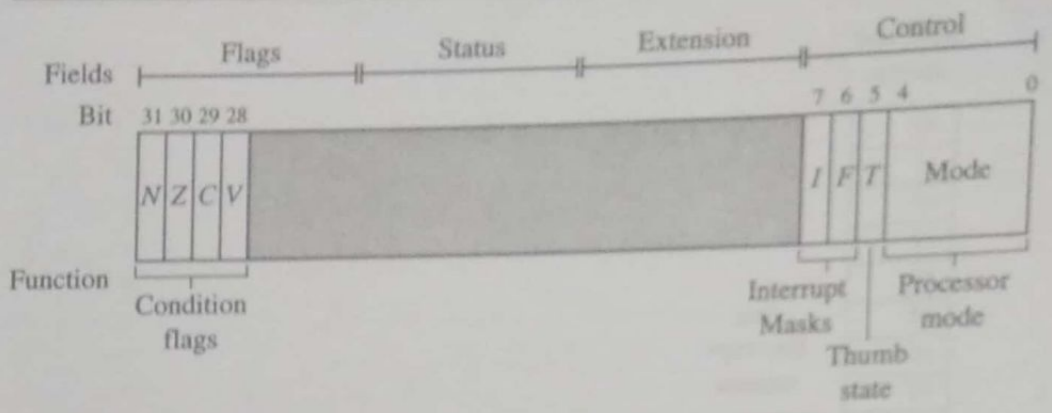


Figure 2.3 A generic program status register (*psr*).

8-bit instructions. We will discuss Jazelle more in Section 2.2.3. It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

For a full description of the *cpsr*, refer to Appendix B.

2.2.1 Processor Modes

The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or nonprivileged: A privileged mode allows full read-write access to the *cpsr*. Conversely, a nonprivileged mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags.

There are seven processor modes in total: six privileged modes (*abort*, *fast interrupt request*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

The processor enters *abort* mode when there is a failed attempt to access memory. *Fast interrupt request* and *interrupt request* modes correspond to the two interrupt levels available on the ARM processor. *Supervisor* mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in. *System* mode is a special version of *user* mode that allows full read-write access to the *cpsr*. *Undefined* mode is used when the processor encounters an instruction that is undefined or not supported by the implementation. *User* mode is used for programs and applications.

2.2.2 Banked Registers

Figure 2.4 shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers* and are identified by the shading in the diagram. They are available only when the processor is in a particular

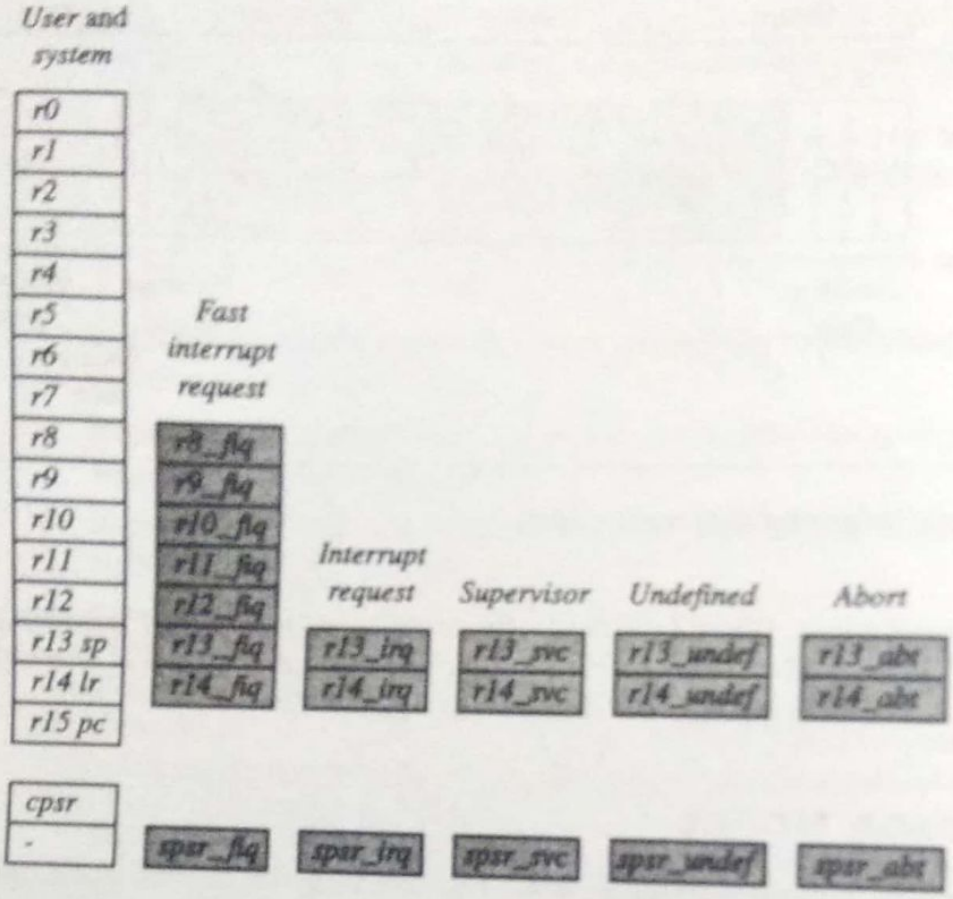


Figure 2.4 Complete ARM register set.

mode; for example, *abort* mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*. Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or *_mode*.

Every processor mode except *user* mode can change mode by writing directly to the mode bits of the *cpsr*. All processor modes except *system* mode have a set of associated banked registers that are a subset of the main 16 registers. A banked register maps one-to-one onto a *user* mode register. If you change processor mode, a banked register from the new mode will replace an existing register.

For example, when the processor is in the *interrupt request* mode, the instructions you execute still access registers named *r13* and *r14*. However, these registers are the banked registers *r13_irq* and *r14_irq*. The *user* mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers. A program still has normal access to the other registers *r0* to *r12*.

The processor mode can be changed by a program that writes directly to the *cpsr* (the processor core has to be in privileged mode) or by hardware when the core responds to

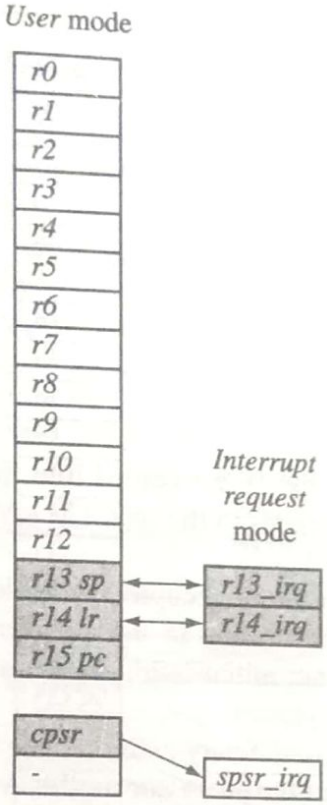


Figure 2.5 Changing mode on an exception.

an exception or interrupt. The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*. Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.

Figure 2.5 illustrates what happens when an interrupt forces a mode change. The figure shows the core changing from *user* mode to *interrupt request* mode, which happens when an *interrupt request* occurs due to an external device raising an interrupt to the processor core. This change causes *user* registers *r13* and *r14* to be banked. The *user* registers are replaced with registers *r13_irq* and *r14_irq*, respectively. Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for *interrupt request* mode.

Figure 2.5 also shows a new register appearing in *interrupt request* mode: the saved program status register *(spsr)*, which stores the previous mode's *cpsr*. You can see in the diagram the *cpsr* being copied into *spsr_irq*. To return back to *user* mode, a special return instruction is used that instructs the core to restore the original *cpsr* from the *spsr_irq* and bank in the *user* registers *r13* and *r14*. Note that the *spsr* can only be modified and read in a privileged mode. There is no *spsr* available in *user* mode.

Table 2.1 Processor mode.

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

Another important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.

Figure 2.3 shows that the current active processor mode occupies the five least significant bits of the *cpsr*. When power is applied to the core, it starts in *supervisor* mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the *cpsr* to set up the stacks for each of the other modes.

Table 2.1 lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*.

2.2.3 State and Instruction Sets

The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb, and Jazelle. The ARM instruction set is only active when the processor is in ARM state. Similarly the Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions. You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.

The Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor. When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor. When the T bit is 1, then the processor is in Thumb state. To change states the core executes a specialized branch instruction. Table 2.2 compares the ARM and Thumb instruction set features.

The ARM designers introduced a third instruction set called *Jazelle*. *Jazelle* executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java bytecodes.

To execute Java bytecodes, you require the Jazelle technology plus a specially modified version of the Java virtual machine. It is important to note that the hardware portion of Jazelle only supports a subset of the Java bytecodes; the rest are emulated in software.

Table 2.2 ARM and Thumb instruction set features.

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

[a] See Section 2.2.6.

Table 2.3 Jazelle instruction set features.

	Jazelle (*cpsr* $T = 0, J = 1$)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

The Jazelle instruction set is a closed instruction set and is not openly available. Table 2.3 gives the Jazelle instruction set features.

2.2.4 Interrupt Masks

Interrupt masks are used to stop specific interrupt requests from interrupting the processor. There are two interrupt request levels available on the ARM processor core—*interrupt request* (IRQ) and *fast interrupt request* (FIQ).

The *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively. The *I* bit masks IRQ when set to binary 1, and similarly the *F* bit masks FIQ when set to binary 1.

2.2.5 Condition Flags

Condition flags are updated by comparisons and the result of ALU operations that specify the S instruction suffix. For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* flag in the *cpsr* is set. This particular subtract instruction specifically updates the *cpsr*.

Table 2.4 Condition flags.

Flag	Flag name	Set when
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero, frequently used to indicate equality
N	Negative	bit 31 of the result is a binary 1

With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction. The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr* directly.

In Jazelle-enabled processors, the *J* bit reflects the state of the core; if it is set, the core is in Jazelle state. The *J* bit is not generally usable and is only available on some processor cores. To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.

Most ARM instructions can be executed conditionally on the value of the condition flags. Table 2.4 lists the condition flags and a short description on what causes them to be set. These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution.

Figure 2.6 shows a typical value for the *cpsr* with both DSP extensions and Jazelle. In this book we use a notation that presents the *cpsr* data in a more human readable form. When a bit is a binary 1 we use a capital letter; when a bit is a binary 0, we use a lowercase letter. For the condition flags a capital letter shows that the flag has been set. For interrupts a capital letter shows that an interrupt is disabled.

In the *cpsr* example shown in Figure 2.6, the *C* flag is the only condition flag set. The rest *nzvq* flags are all clear. The processor is in ARM state because neither the Jazelle *j* or Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled. Finally, you

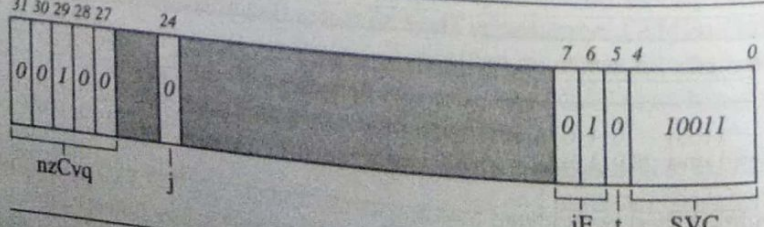


Figure 2.6 Example: *cpsr* = nzCvqjiFt_SVC.

Table 2.5 Condition mnemoni

Mnemonic
EQ
NE
CS HS
CC LO
MI
PL
VS
VC
HI
LS
GE
LT
GT
LE
AL

can see from
equal to bina

2.2.6 CONDITION

Conditiona
Most instr
based on t
condition
is executed
The c
into the i
condition

2.3 PIPELI

A pipeli
speeds
decode
assemb

Table 2.5 Condition mnemonics.

Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

can see from the figure the processor is in *supervisor* (*SVC*) mode since the mode[4:0] is equal to binary 10011.

2.2.6 Conditional Execution

Conditional execution controls whether or not the core will execute an instruction. Most instructions have a condition attribute that determines if the core will execute it based on the setting of the condition flags. Prior to execution, the processor compares the condition attribute with the condition flags in the *cpsr*. If they match, then the instruction is executed; otherwise the instruction is ignored.

The condition attribute is postfixed to the instruction mnemonic, which is encoded into the instruction. Table 2.5 lists the conditional execution code mnemonics. When a condition mnemonic is not present, the default behavior is to set it to always (*AL*) execute.

2.3 Pipeline

A pipeline is the mechanism a RISC processor uses to execute instructions. Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed. One way to view the pipeline is to think of it as an automobile assembly line, with each stage carrying out a particular task to manufacture the vehicle.

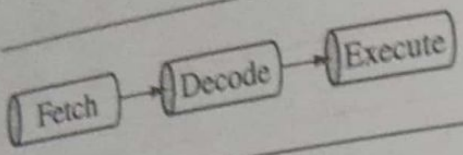


Figure 2.7 ARM7 Three-stage pipeline.

Figure 2.7 shows a three-stage pipeline:

- *Fetch* loads an instruction from memory.
- *Decode* identifies the instruction to be executed.
- *Execute* processes the instruction and writes the result back to a register.

Figure 2.8 illustrates the pipeline using a simple example. It shows a sequence of three instructions being fetched, decoded, and executed by the processor. Each instruction takes a single cycle to complete after the pipeline is filled.

The three instructions are placed into the pipeline sequentially. In the first cycle the core fetches the ADD instruction from memory. In the second cycle the core fetches the SUB instruction and decodes the ADD instruction. In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched. This procedure is called *filling the pipeline*. The pipeline allows the core to execute an instruction every cycle.

As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn increases the performance. The system *latency* also increases because it takes more cycles to fill the pipeline before the core can execute an instruction. The increased pipeline length also means there can be data dependency between certain stages. You can write code to reduce this dependency by using *instruction scheduling* (for more information on instruction scheduling take a look at Chapter 6).

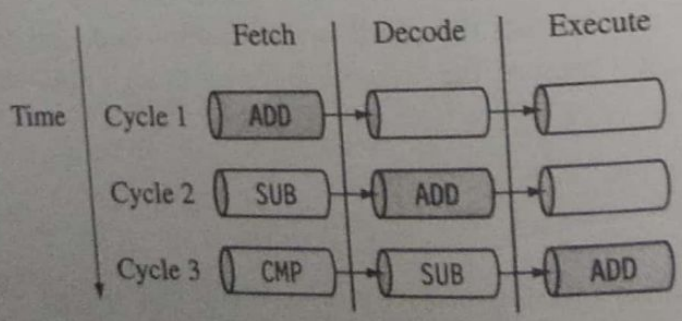


Figure 2.8 Pipelined instruction sequence.

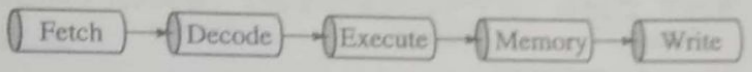


Figure 2.9 ARM9 five-stage pipeline.

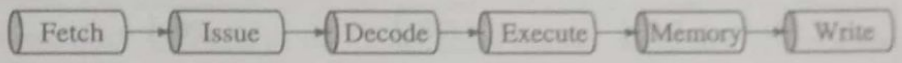


Figure 2.10 ARM10 six-stage pipeline.

The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure 2.9. The ARM9 adds a memory and writeback stage, which allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz—an increase in instruction throughput by around 13% compared with an ARM7. The maximum core frequency attainable using an ARM9 is also higher.

The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in Figure 2.10. The ARM10 can process on average 1.3 Dhrystone MIPS per MHz, about 34% more throughput than an ARM7 processor core, but again at a higher latency cost.

Even though the ARM9 and ARM10 pipelines are different, they still use the same *pipeline executing characteristics* as an ARM7. Code written for the ARM7 will execute on an ARM9 or ARM10.

2.3.1 Pipeline Executing Characteristics

The ARM pipeline has not processed an instruction until it passes completely through the execute stage. For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.

Figure 2.11 shows an instruction sequence on an ARM7 pipeline. The `MSR` instruction is used to enable IRQ interrupts, which only occurs once the `MSR` instruction completes the execute stage of the pipeline. It clears the *I* bit in the *cpsr* to enable the IRQ interrupts. Once the `ADD` instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.

Figure 2.12 illustrates the use of the pipeline and the program counter *pc*. In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead. This is important when the *pc* is used for calculating a relative offset and is an

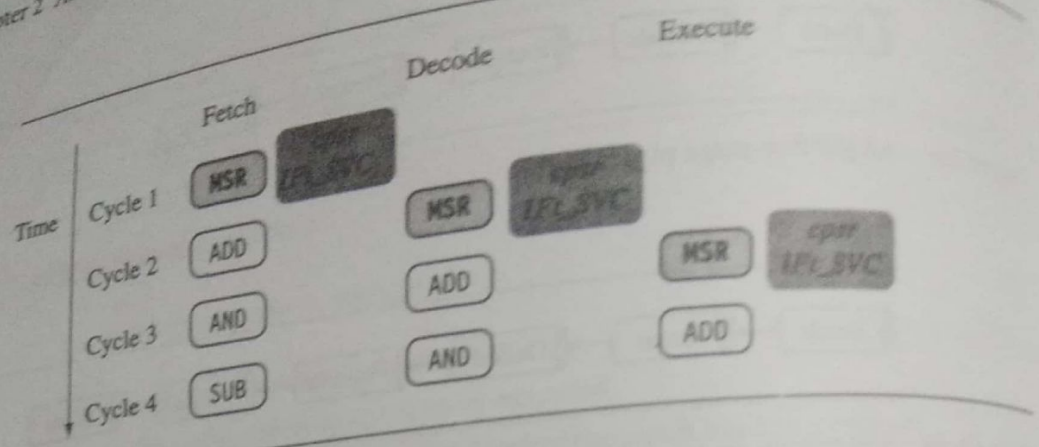


Figure 2.11 ARM instruction sequence.

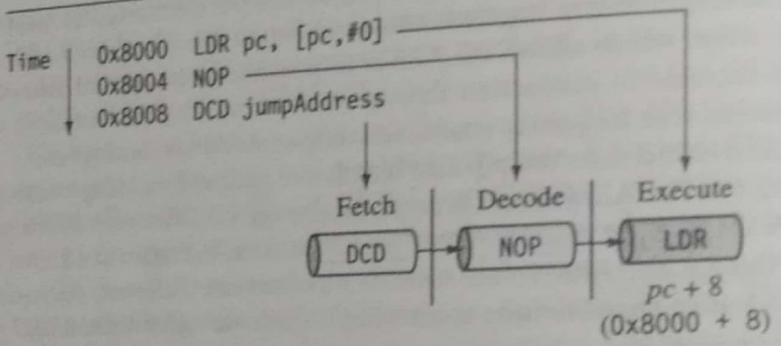


Figure 2.12 Example: *pc* = address + 8.

architectural characteristic across all the pipelines. Note when the processor is in Thumb state the *pc* is the instruction address plus 4.

There are three other characteristics of the pipeline worth mentioning. First, the execution of a branch instruction or branching by the direct modification of the *pc* causes the ARM core to flush its pipeline.

Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.

Third, an instruction in the execute stage will complete even though an interrupt has been raised. Other instructions in the pipeline will be abandoned, and the processor will start filling the pipeline from the appropriate entry in the vector table.

2.4 Exceptions, Interrupts, and the Vector Table

When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.

The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher address in memory (starting at the offset 0xffff0000). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

- *Reset vector* is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- *Undefined instruction vector* is used when the processor cannot decode an instruction.
- *Software interrupt vector* is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
- *Prefetch abort vector* occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
- *Data abort vector* is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.
- *Interrupt request vector* is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.

Table 2.6 The vector table.

Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

- *Fast interrupt request vector* is similar to the interrupt request but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the cpsr.

2.5 CORE EXTENSIONS

The hardware extensions covered in this section are standard components placed next to the ARM core. They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications. Each ARM family has different extensions available.

There are three hardware extensions ARM wraps around the core: cache and tightly coupled memory, memory management, and the coprocessor interface.

2.5.1 CACHE AND TIGHTLY COUPLED MEMORY

The cache is a block of fast memory placed between main memory and the core. It allows for more efficient fetches from some memory types. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory. Most ARM-based embedded systems use a single-level cache internal to the processor. Of course, many small embedded systems do not require the performance gains that a cache brings.

ARM has two forms of cache. The first is found attached to the Von Neumann–style cores. It combines both data and instruction into a single unified cache, as shown in Figure 2.13. For simplicity, we have called the glue logic that connects the memory system to the AMBA bus *logic and control.*

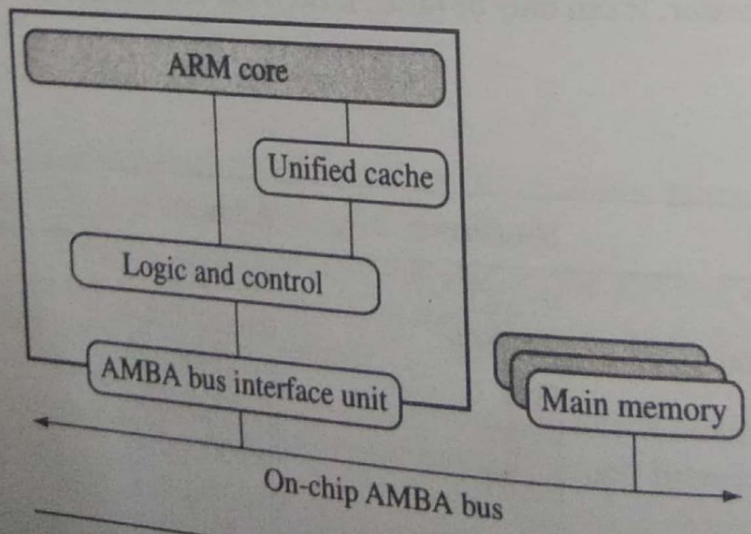


Figure 2.13 A simplified Von Neumann architecture with cache.

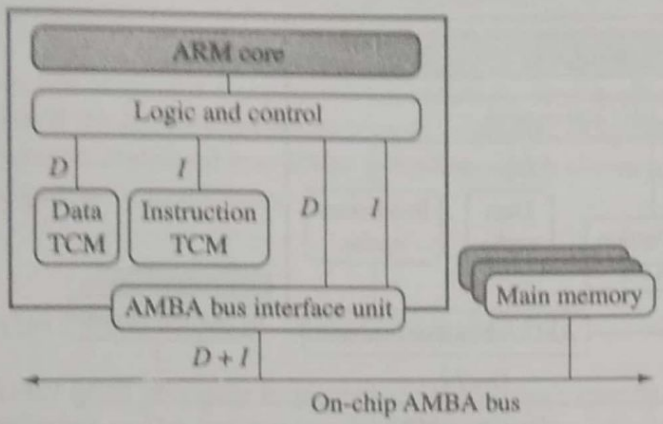


Figure 2.14 A simplified Harvard architecture with TCMs.

By contrast, the second form, attached to the Harvard-style cores, has separate caches for data and instruction.

A cache provides an overall increase in performance but at the expense of predictable execution. But for real-time systems it is paramount that code execution is *deterministic*—the time taken for loading and storing instructions or data must be predictable. This is achieved using a form of memory called *tightly coupled memory* (TCM). TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data—critical for real-time algorithms requiring deterministic behavior. TCMs appear as memory in the address map and can be accessed as fast memory. An example of a processor with TCMs is shown in Figure 2.14.

By combining both technologies, ARM processors can have both improved performance and predictable real-time response. Figure 2.15 shows an example core with a combination of caches and TCMs.

2.5.2 MEMORY MANAGEMENT

Embedded systems often use multiple memory devices. It is usually necessary to have a method to help organize these devices and protect the system from applications trying to make inappropriate accesses to hardware. This is achieved with the assistance of memory management hardware.

ARM cores have three different types of memory management hardware—no extensions providing no protection, a memory protection unit (MPU) providing limited protection, and a memory management unit (MMU) providing full protection:

- *Nonprotected memory* is fixed and provides very little flexibility. It is normally used for small, simple embedded systems that require no protection from rogue applications.

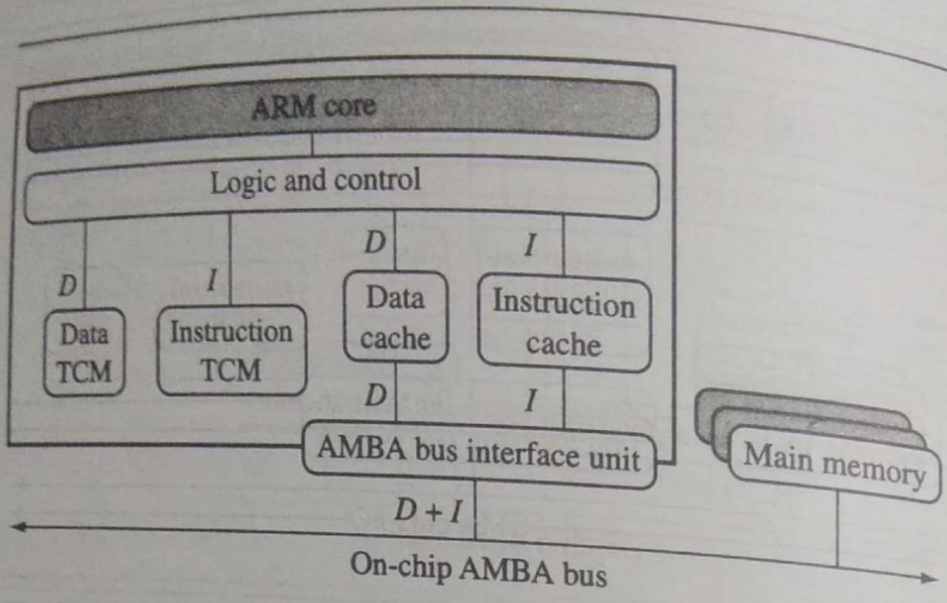


Figure 2.15 A simplified Harvard architecture with caches and TCMs.

- *MPUs* employ a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permissions. This type of memory management is used for systems that require memory protection but don't have a complex memory map. The MPU is explained in Chapter 13.
- *MMUs* are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory. These tables are stored in main memory and provide a virtual-to-physical address map as well as access permissions. MMUs are designed for more sophisticated platform operating systems that support multitasking. The MMU is explained in Chapter 14.

2.5.3 COPROCESSORS

Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers. More than one coprocessor can be added to the ARM core via the coprocessor interface.

The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface. Consider, for example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.

The coprocessor can also extend the instruction set by providing a specialized group of new instructions. For example, there are a set of specialized instructions that can

be added to the standard ARM instruction set to process vector floating-point (VFP) operations.

These new instructions are processed in the decode stage of the ARM pipeline. If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor. But if the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.

2.6 Architecture Revisions

Every ARM processor implementation executes a specific *instruction set architecture* (ISA), although an ISA revision may have more than one processor implementation.

The ISA has evolved to keep up with the demands of the embedded market. This evolution has been carefully managed by ARM, so that code written to execute on an earlier architecture revision will also execute on a later revision of the architecture.

Before we go on to explain the evolution of the architecture, we must introduce the ARM processor nomenclature. The nomenclature identifies individual processors and provides basic information about the feature set.

2.6.1 Nomenclature

ARM uses the nomenclature shown in Figure 2.16 to describe the processor implementations. The letters and numbers after the word "ARM" indicate the features a processor

---

ARM{x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{-S}

x—family
y—memory management/protection unit
z—cache
T—Thumb 16-bit decoder
D—JTAG debug
M—fast multiplier
I—EmbeddedICE macrocell
E—enhanced instructions (assumes TDMI)
J—Jazelle
F—vector floating-point unit
S—synthesizible version

Figure 2.16 ARM nomenclature.

may have. In the future the number and letter combinations may change as more features are added. Note the nomenclature does not include the architecture revision information. There are a few additional points to make about the ARM nomenclature:

- All ARM cores after the ARM7TDMI include the *TDMI* features even though they may not include those letters after the "ARM" label.
- The processor *family* is a group of processor implementations that share the same hardware characteristics. For example, the ARM7TDMI, ARM740T, and ARM720T all share the same family characteristics and belong to the ARM7 family.
- *JTAG* is described by IEEE 1149.1 Standard Test Access Port and boundary scan architecture. It is a serial protocol used by ARM to send and receive debug information between the processor core and test equipment.
- *EmbeddedICE macrocell* is the debug hardware built into the processor that allows breakpoints and watchpoints to be set.
- *Synthesizable* means that the processor core is supplied as source code that can be compiled into a form easily used by EDA tools.

2.6.2 Architecture Evolution

The architecture has continued to evolve since the first ARM processor implementation was introduced in 1985. Table 2.7 shows the significant architecture enhancements from the original architecture version 1 to the current version 6 architecture. One of the most significant changes to the ISA was the introduction of the Thumb instruction set in ARMv4T (the ARM7TDMI processor).

Table 2.8 summarizes the various parts of the program status register and the availability of certain features on particular instruction architectures. "All" refers to the ARMv4 architecture and above.

2.7 ARM Processor Families

ARM has designed a number of processors that are grouped into different families according to the core they use. The families are based on the ARM7, ARM9, ARM10, and ARM11 cores. The postfix numbers 7, 9, 10, and 11 indicate different core designs. The ascending number equates to an increase in performance and sophistication. ARM8 was developed but was soon superseded.

Table 2.9 shows a rough comparison of attributes between the ARM7, ARM9, ARM10, and ARM11 cores. The numbers quoted can vary greatly and are directly dependent upon the type and geometry of the manufacturing process, which has a direct effect on the frequency (MHz) and power consumption (watts).

Table 2.7 Revision history.

Revision	Example core implementation	ISA enhancement
ARMv1	ARM1	First ARM processor
ARMv2	ARM2	26-bit addressing 32-bit multiplier
ARMv2a	ARM3	32-bit coprocessor support On-chip cache Atomic swap instruction
ARMv3	ARM6 and ARM7DI	Coprocessor 15 for cache management 32-bit addressing Separate *cpsr* and *spsr* New modes—*undefined instruction* and *abort* MMU support—virtual memory
ARMv3M	ARM7M	Signed and unsigned long multiply instructions
ARMv4	StrongARM	Load-store instructions for signed and unsigned halfwords/bytes New mode—*system* Reserve SWI space for architecturally defined operations 26-bit addressing mode no longer supported
ARMv4T	ARM7TDMI and ARM9T	Thumb
ARMv5TE	ARM9E and ARM10E	Superset of the ARMv4T Extra instructions added for changing state between ARM and Thumb Enhanced multiply instructions Extra DSP-type instructions Faster multiply accumulate
ARMv5TEJ	ARM7EJ and ARM926EJ	Java acceleration
ARMv6	ARM11	Improved multiprocessor instructions Unaligned and mixed endian data handling New multimedia instructions

Within each ARM family, there are a number of variations of memory management, cache, and TCM processor extensions. ARM continues to expand both the number of families available and the different variations within each family.

You can find other processors that execute the ARM ISA such as StrongARM and XScale. These processors are unique to a particular semiconductor company, in this case Intel.

Table 2.10 summarizes the different features of the various processors. The next subsections describe the ARM families in more detail, starting with the ARM7 family.

Table 2.8 Description of the *cpsr*.

Parts	Bits	Architectures	Description
Mode	4:0	all	processor mode
T	5	ARMv4T	Thumb state
I & *F*	7:6	all	interrupt masks
J	24	ARMv5TEJ	Jazelle state
Q	27	ARMv5TE	condition flag
V	28	all	condition flag
C	29	all	condition flag
Z	30	all	condition flag
N	31	all	condition flag

Table 2.9 ARM family attribute comparison.

	ARM7	ARM9	ARM10	ARM11
Pipeline depth	three-stage	five-stage	six-stage	eight-stage
Typical MHz	80	150	260	335
mW/MHz[a]	0.06 mW/MHz	0.19 mW/MHz (+ cache)	0.5 mW/MHz (+ cache)	0.4 mW/MHz (+ cache)
MIPS[b]/MHz	0.97	1.1	1.3	1.2
Architecture	Von Neumann	Harvard	Harvard	Harvard
Multiplier	8 × 32	8 × 32	16 × 32	16 × 32

[a] Watts/MHz on the same 0.13 micron process.
[b] MIPS are Dhrystone VAX MIPS.

2.7.1 ARM7 Family

The ARM7 core has a Von Neumann–style architecture, where both data and instructions use the same bus. The core has a three-stage pipeline and executes the architecture ARMv4T instruction set.

The ARM7TDMI was the first of a new range of processors introduced in 1995 by ARM. It is currently a very popular core and is used in many 32-bit embedded processors. It provides a very good performance-to-power ratio. The ARM7TDMI processor core has been licensed by many of the top semiconductor companies around the world and is the first core to include the Thumb instruction set, a fast multiply instruction, and the EmbeddedICE debug technology.

Table 2.10 ARM processor variants.

CPU core	MMU/MPU	Cache	Jazelle	Thumb	ISA	E[a]
ARM7TDMI	none	none	no	yes	v4T	no
ARM7EJ-S	none	none	yes	yes	v5TEJ	yes
ARM720T	MMU	unified—8K cache	no	yes	v4T	no
ARM920T	MMU	separate—16K /16K $D+I$ cache	no	yes	v4T	no
ARM922T	MMU	separate—8K/8K $D+I$ cache	no	yes	v4T	no
ARM926EJ-S	MMU	separate—cache and TCMs configurable	yes	yes	v5TEJ	yes
ARM940T	MPU	separate—4K/4K $D+I$ cache	no	yes	v4T	no
ARM946E-S	MPU	separate—cache and TCMs configurable	no	yes	v5TE	yes
ARM966E-S	none	separate—TCMs configurable	no	yes	v5TE	yes
ARM1020E	MMU	separate—32K/32K $D+I$ cache	no	yes	v5TE	yes
ARM1022E	MMU	separate—16K/16K $D+I$ cache	no	yes	v5TE	yes
ARM1026EJ-S	MMU and MPU	separate—cache and TCMs configurable	yes	yes	v5TE	yes
ARM1136J-S	MMU	separate—cache and TCMs configurable	yes	yes	v6	yes
ARM1136JF-S	MMU	separate—cache and TCMs configurable	yes	yes	v6	yes

[a] E extension provides enhanced multiply instructions and saturation.

One significant variation in the ARM7 family is the ARM7TDMI-S. The ARM7TDMI-S has the same operating characteristics as a standard ARM7TDMI but is also synthesizable.

ARM720T is the most flexible member of the ARM7 family because it includes an MMU. The presence of the MMU means the ARM720T is capable of handling the Linux and Microsoft embedded platform operating systems. The processor also includes a unified 8K cache. The vector table can be relocated to a higher address by setting a coprocessor 15 register.

Another variation is the ARM7EJ-S processor, also synthesizable. ARM7EJ-S is quite different since it includes a five-stage pipeline and executes ARMv5TEJ instructions. This version of the ARM7 is the only one that provides both Java acceleration and the enhanced instructions but without any memory protection.

2.7.2 ARM9 FAMILY

The ARM9 family was announced in 1997. Because of its five-stage pipeline, the ARM9 processor can run at higher clock frequencies than the ARM7 family. The extra stages improve the overall performance of the processor. The memory system has been redesigned to follow the Harvard architecture, which separates the data D and instruction I buses.

The first processor in the ARM9 family was the ARM920T, which includes a separate $D + I$ cache and an MMU. This processor can be used by operating systems requiring virtual memory support. ARM922T is a variation on the ARM920T but with half the $D + I$ cache size.

The ARM940T includes a smaller $D + I$ cache and an MPU. The ARM940T is designed for applications that do not require a platform operating system. Both ARM920T and ARM940T execute the architecture v4T instructions.

The next processors in the ARM9 family were based on the ARM9E-S core. This core is a synthesizable version of the ARM9 core with the E extensions. There are two variations: the ARM946E-S and the ARM966E-S. Both execute architecture v5TE instructions. They also support the optional *embedded trace macrocell* (ETM), which allows a developer to trace instruction and data execution in real time on the processor. This is important when debugging applications with time-critical segments.

The ARM946E-S includes TCM, cache, and an MPU. The sizes of the TCM and caches are configurable. This processor is designed for use in embedded control applications that require deterministic real-time response. In contrast, the ARM966E does not have the MPU and cache extensions but does have configurable TCMs.

The latest core in the ARM9 product line is the ARM926EJ-S synthesizable processor core, announced in 2000. It is designed for use in small portable Java-enabled devices such as 3G phones and personal digital assistants (PDAs). The ARM926EJ-S is the first ARM processor core to include the Jazelle technology, which accelerates Java bytecode execution. It features an MMU, configurable TCMs, and $D + I$ caches with zero or nonzero wait state memories.

2.7.3 ARM10 FAMILY

The ARM10, announced in 1999, was designed for performance. It extends the ARM9 pipeline to six stages. It also supports an optional vector floating-point (VFP) unit, which adds a seventh stage to the ARM10 pipeline. The VFP significantly increases floating-point performance and is compliant with the IEEE 754.1985 floating-point standard.

The ARM1020E is the first processor to use an ARM10E core. Like the ARM9E, it includes the enhanced E instructions. It has separate 32K $D + I$ caches, optional vector floating-point unit, and an MMU. The ARM1020E also has a dual 64-bit bus interface for increased performance.

ARM1026EJ-S is very similar to the ARM926EJ-S but with both MPU and MMU. This processor has the performance of the ARM10 with the flexibility of an ARM926EJ-S.

2.7.4 ARM11 Family

The ARM1136J-S, announced in 2003, was designed for high performance and power-efficient applications. ARM1136J-S was the first processor implementation to execute architecture ARMv6 instructions. It incorporates an eight-stage pipeline with separate load-store and arithmetic pipelines. Included in the ARMv6 instructions are single instruction multiple data (SIMD) extensions for media processing, specifically designed to increase video processing performance.

The ARM1136JF-S is an ARM1136J-S with the addition of the vector floating-point unit for fast floating-point operations.

2.7.5 Specialized Processors

StrongARM was originally co-developed by Digital Semiconductor and is now exclusively licensed by Intel Corporation. It is has been popular for PDAs and applications that require performance with low power consumption. It is a Harvard architecture with separate $D + I$ caches. StrongARM was the first high-performance ARM processor to include a five-stage pipeline, but it does not support the Thumb instruction set.

Intel's XScale is a follow-on product to the StrongARM and offers dramatic increases in performance. At the time of writing, XScale was quoted as being able to run up to 1 GHz. XScale executes architecture v5TE instructions. It is a Harvard architecture and is similar to the StrongARM, as it also includes an MMU.

SC100 is at the other end of the performance spectrum. It is designed specifically for low-power security applications. The SC100 is the first SecurCore and is based on an ARM7TDMI core with an MPU. This core is small and has low voltage and current requirements, which makes it attractive for smart card applications.

2.8 Summary

In this chapter we focused on the hardware fundamentals of the actual ARM processor. The ARM processor can be abstracted into eight components—ALU, barrel shifter, MAC, register file, instruction decoder, address register, incrementer, and sign extend.

ARM has three instruction sets—ARM, Thumb, and Jazelle. The register file contains 37 registers, but only 17 or 18 registers are accessible at any point in time; the rest are banked according to processor mode. The current processor mode is stored in the *cpsr*. It holds the current status of the processor core as well interrupt masks, condition flags, and state. The state determines which instruction set is being executed.

An ARM processor comprises a core plus the surrounding components that interface it with a bus. The core extensions include the following:

- *Caches* are used to improve the overall system performance.
- *TCMs* are used to improve deterministic real-time response.

- *Memory management* is used to organize memory and protect system resources.
- *Coprocessors* are used to extend the instruction set and functionality. Coprocessor 15 controls the cache, TCMs, and memory management.

An ARM processor is an implementation of a specific instruction set architecture (ISA). The ISA has been continuously improved from the first ARM processor design. Processors are grouped into implementation families (ARM7, ARM9, ARM10, and ARM11) with similar characteristics.

Chapter 9

Embedded Systems Design and Development

THINGS TO LOOK FOR . . .

- Things to consider in a design.
- The product life cycle.
- The five steps to design.
- The need to understand the environment and the system being designed.
- The difference between requirements definition and specification.
- Motivation for and objective when partitioning a system.
- Coupling and cohesion and why they are important.
- The differences between functional and architectural models of a system.
- Motivation for and timing of static and dynamic analysis of a design.
- Capitalization and reuse of designs.
- Requirements traceability.

9.0 INTRODUCTION

In this chapter, we will study the major phases of the development process for embedded systems. The more detailed aspects of that process will be explored in conjunction with the design and test of the specific hardware and software elements of the system.

We will learn that design is the process of translating customer requirements into a working system and that the complexity of contemporary systems demands a formal approach and formal methods. Working from a formal specification of a problem, we will look at ways of partitioning the system as a prelude to developing a functional design. We will then examine the process of mapping a functional model on to an architectural structure and ultimately to a working prototype. To help ensure the robustness of the ultimate product, we will illustrate how to critically analyze the design both during and after development.

We will also examine several other important considerations in the design life cycle, including intellectual property, component/module reuse, and requirements management and the archival process.

As we begin to think about a new product or as we add new features to an existing one, we must look at things from many different points of view. The most important of these perspectives is the customer's since he or she finances the development of the product either directly through an agreed upon contract or indirectly through a purchase. The best design

is of little value if no one is willing to buy it. So, we pose the question: What kinds of things should be considered?

costs, features
real, perceived needs

If we look at products, we must know how to measure *costs* and *features*. We must be able to identify and distinguish between *real* and *perceived needs*. Too often when we talk with customers about new products, the essential "requirement" in the next generation product is that which was missing when a problem arose this morning.

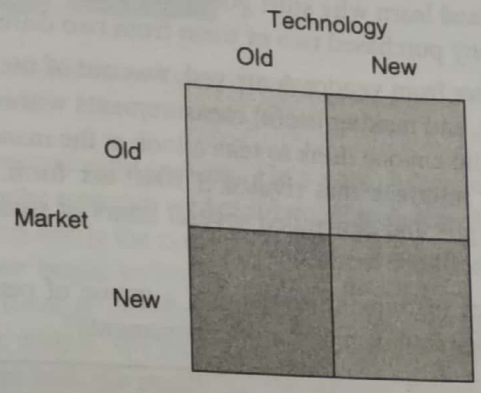


Figure 9.0 Market–Technology Trade-off

old technology
old markets

It is important to learn how to make market and technology trade-offs. Several years ago the very simple table shown in Figure 9.0 was proposed. Taking *old technology* into *old markets* is a reasonable and safe strategy. These are the niche markets and often provide support and evolutionary growth for products that are no longer in a vendor's mainstream offering.

new technology, new markets

Taking *new technology* into *new markets* is difficult and risky. At the same time, the rewards can be very high. The personal computer is a very good example. Xerox and Apple both had limited success with their early offerings. The people and the full technology were simply not ready.

new, existing

Taking *new* technology in to an *existing* area or *existing* technology in to a *new* area is easier. At least one portion of the problem—the market or the technology—is well understood and well developed.

deadlines, costs

We must understand the importance of *deadlines* and *costs*. Product development is based on a (directly or indirectly) negotiated contract between us and the customer(s). Failure to respect development and delivery costs or schedules leads to loss of sales, market share, and credibility.

reliability, safety, quality

We also must always consider *reliability*, *safety*, and *quality* in the products we design. We learned about these in Chapter 8. Beyond an obvious need to work properly, the product must be *robust*. Simply put, "Does it do what it's supposed to?" and "How does it behave with unexpected inputs?" Robust means much more than this, however. Robust also implies that the system performs even if it is partially damaged, or under extreme temperature conditions, or if it is dropped. If a product does what it is supposed to do but is fragile and buggy, the product is not robust.

robust

documentation
post-sales support

The *documentation* we produce to accompany the product must be clear and understandable. The product must be easy to use—intuitive rather than counterintuitive. *Post-sales support*, including the correction of bugs, is very important. Lack of quality has two costs. The first is obvious and immediate—the cost to repair, which is often small. The second is a hidden cost—the loss of customer confidence and sales—and it can be very large. Once confidence lost, it is very difficult to regain.

A Simple Example

Years ago, when developing some of the early microprocessor-based embedded systems, we would encounter problems as we debugged the hardware and software. At that time, tools were few and far between. This was a new field.

One very powerful tool for helping to track down such problems is called a logic analyzer. It allows one to follow which instructions the processor is executing (in real time) and learn why stuff goes in and never comes out. We had to have one, so, our company purchased two of them from two different vendors.

The analyzer from vendor A arrived, was out of the box, on the bench, connected to the system, and making useful measurements within 10 to 15 minutes. Only several days later did anyone think to take a look at the manual. The analyzer from vendor B had a user interface that rivaled a 1040 tax form. Its one-inch thick manual was equally cryptic and demanded several hours of study before even the simplest measurements could be made.

Guess which instrument always has a queue of people waiting to use it and guess which vendor sold us many more instruments?

9.1 SYSTEM DESIGN AND DEVELOPMENT

System design and development is a challenging problem. What makes it fun and exciting is its very large creative component. There are no rules, no steps to follow to make one creative. There are, however, a large collection of rules to ensure the opposite. Consider a new child. Each comes into this world, eyes wide open with a million questions. Why is the sky blue? Why is the sun yellow? Why can't we see the air? Where does air come from anyway? What do we do? We put them in school. We teach them the rules. Walk into any group of little ones and ask, how many of you can sing? How many of you can draw? Almost every tiny hand leaps up. Go into any similar group of adults and ask the same questions. Everyone is suddenly fascinated with their shoes. One hand may slowly come up. Why? We place too many restrictions on our thinking. Sure, we may need 10 million dollars worth of electronic equipment to give our voice perfect pitch, but so what. We need to remove artificial restrictions that we impose on our thinking.

Look at the little ones drawing or coloring. What do we tell them? No, people aren't purple. Cows can't fly. Fish don't have legs—anymore. Oh, and by the way, always color in the lines. . . . and let's also learn how to be creative.

9.1.1 Getting Ready—Start Thinking

Okay, let's start. Driving is always a good place to begin. The rules are easy. Keep the yellow line on your left and the white on your right—except in Britain and several other places. Now the chance to be creative. In the autumn in the northern parts of the world, the days are warm, but the nights start getting colder. Often there is a bit of fog that makes an appearance as well. By the morning, the fog and chill have combined to give a very fine glaze of ice on the road. We call this black ice; it gives us the opportunity to be creative. Hop in the car and race out onto the road. What's this nonsense about staying in the lines?

Now that perhaps we have decided that maybe we can be just a little creative, let's begin to explore. As we begin thinking about a new design, we will discover that there are a lot of things to be considered. The problem may not always be what it seems at first blush. Roger van Oech in *A Whack on the Side of the Head*, Warner Brooks (1983), says "Always

A Whack on the Side of the Head

Figure 9.1 Old Woman—Young Woman

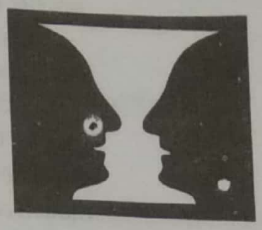


Figure 9.2 Goblet or Statue

look for the second right answer." He's right. As we begin, it is important to understand the problem to make sure that we are solving the right problem. Consider the illustration in Figure 9.1. Which one is the correct image? Is it the old lady or the young one?

When we begin trying to solve a problem, it is important to talk with everyone involved; to listen to different opinions; to see how the design might affect the people who have to work with it. We have to take the time to look at different views of the problem, to look at it from both the inside and the outside. Based on our view, we can have a couple of different interpretations of the problem presented in Figure 9.2. Are we building a goblet, or are we building two statues?

There will always be occasions in which we have too much information, too many opinions, or too many details. Remember the old expression of not being able to see the forest for the trees. The same holds true as we begin trying to understand a problem during the early stages of a design. Look at this next drawing in Figure 9.3. What do you see? This interesting design looks perhaps like a snowflake. This is a case in which we have too much information.

Let's remove some of the information as in Figure 9.4; if we take a more abstract view of the problem, the solution is easier to see.

Now that we have a start, let's look at the design problem. Let's look at each design as a chance to explore.

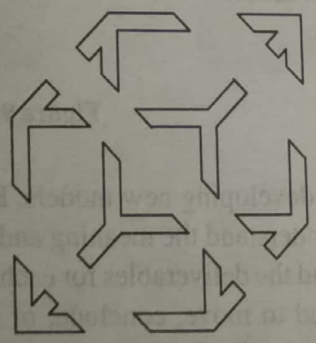


Figure 9.3 Too Much Information

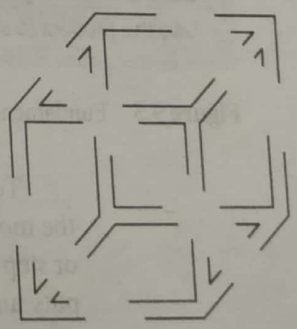


Figure 9.4 Remove Some Information

9.1.2 Getting Started

Designing and developing embedded systems does raise some interesting challenges and does require a large number of decisions. Some of those decisions require knowledge about the problem, others require knowledge about the tools and techniques that may be available, and still others choose methods for approaching the solution. There will often be still more

things to think about that are not related to the technical part of the problem at all. The collection of things we do as we move from requirement to application is often called the *product life cycle*.

product life cycle

Like so many other things in life, there are probably as many different product life-cycle models as there are people designing these systems. Who said there isn't any creativity? Each of these models has its supporters, and each also has its group of detractors. The goal in the next few pages is to introduce some of the more important things one should think about when executing a design, to present several of the more common life-cycle models, and to present some guidelines for things that have worked on successful projects. Despite what they tell you, there are no hard and fast rules—well, perhaps there are a few: learn a lot with each project, have fun, and do the job right, to the best of your ability. Let's get started.

9.2 LIFE-CYCLE MODELS

The product life cycle of an embedded application is purely a descriptive representation. It breaks the development process into a series of interrelated activities. Each activity plays a role of transforming its input (specification) into an output (a selected solution). The steps are organized according to a *design process model—the life-cycle model*. Formality in design provides the structure for a development that lets us creatively explore the design while using the tools to manage some of the more mechanical issues. We use the structure as an aid rather than as something that encumbers design.

design process model, the life-cycle model

As we have commented already, the related literature presents a variety of proposed approaches and models. At the end of the day, all have the same basic goal, however: they all have similar phases. Perhaps we could more accurately say that they all have similar needs or goals or objectives. These needs are very simple, as shown in Figure 9.5.

Several of the historically more common models or approaches are listed in Figure 9.6,

- Find out what the customers want.
- Think of a way to give them what they want.
- Prove what you've done by building and testing it.
- Build a lot of the product to prove that it wasn't an accident.
- Use the product to solve the customer's problem.

Figure 9.5 Fundamentals of Design

- Waterfall
- V Cycle
- Spiral
- Rapid Prototype

Figure 9.6 Common Life-Cycle Models

Today, we are continually developing new models. But whichever model we choose, the most important point is to understand the meaning and intent or objective of each phase or step in the process. Understand the deliverables for each step as well as the necessary outputs and inputs that are required to move, conclude, or enter each phase in the selected model. Then follow those and don't take shortcuts. We will look briefly at each of these four models momentarily. Before we do so, let's look at another model that fits just about any phase of engineering; it looks something like the one shown in Figure 9.7.

hockey stick model

This is called the *hockey stick model* or curve; its shape is strongly suggestive of where the name originated. We have talked about how important it is to address reliability and safety early in the requirements specification and design phases of the life cycle. The hockey stick curve, shown in Figure 9.7, provides an intuitive feel as to why. If we label the horizontal axis as time and the vertical one as cost, and apply it here, we see that the longer

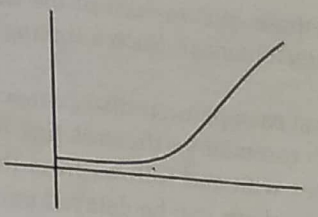


Figure 9.7 The Hockey Stick Curve

we delay in addressing those issues, the greater the cost will be. Cost is not limited to money alone.

Waterfall model

Let's begin with the *Waterfall model*. Use your artistic creativity here. Its name evokes its sound, which evokes the philosophy and approach engendered in the model.

9.2.1 The Waterfall Model

Waterfall model

The *Waterfall model* represents a cycle—specifically, a series of steps appearing much like a waterfall, sequentially, one below the next as we see in Figure 9.8.

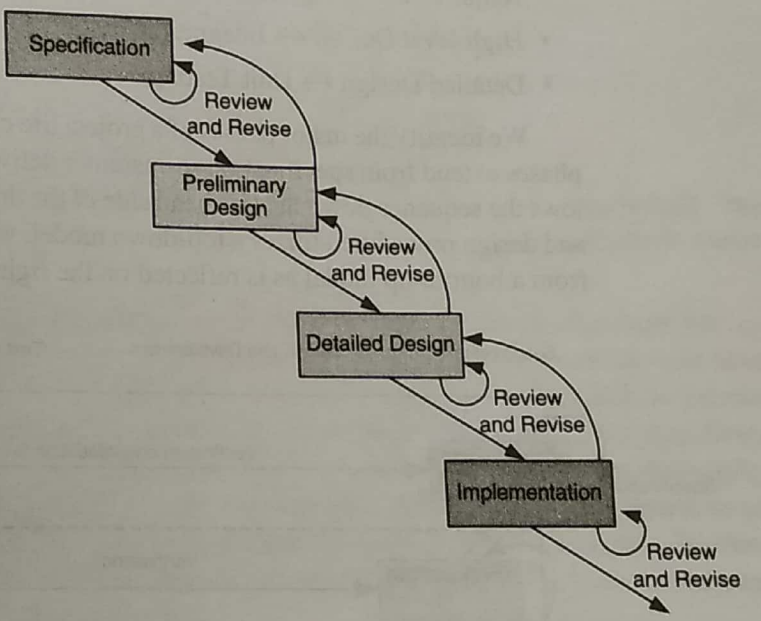


Figure 9.8 The Waterfall Life-Cycle Model

The steps are:

- Specification
- Preliminary design
- Design review
- Detailed design
- Design review
- Implementation
- Review

Complete this phase and go on to the next

Together, these capture each of the needs we identified earlier. Successive steps are linked in a chained manner. Such a linking tends to say: *Complete this phase and go on to the next.*

Observe that each phase is also connected to the previous phase. That reverse connection provides an essential verification link backwards to ensure that the solution (in its current form) agrees with and follows from the specification. With the Waterfall model, the recognition of problems can be delayed until later states of development where the cost of repair is higher (the hockey stick curve). The Waterfall model is limited in the sense that it does not consider the typically iterative nature of real-world design.

9.2.2 The V Cycle Model

V Cycle

The *V Cycle* is similar to the Waterfall model except that it places greater emphasis on the importance of addressing testing activities up front instead of later in the life cycle. Each stage associates the development activity for that phase with a test or validation at the same level. Each test phase is identified with its matching development phase as we see in Figure 9.9.

In the diagram, we have

- Requirements ↔ System/Functional Testing
- High-level Design ↔ Integration Testing
- Detailed Design ↔ Unit Testing

We identify the major phases of a project life cycle across the top of the drawing. These phases extend from specification to customer delivery and postdelivery support. If one follows the sequence down the left-hand side of the drawing, one can see that the specification and design procedure utilizes a top-down model, whereas implementation and test proceed from a bottom-up model as is reflected on the right-hand side of the drawing.

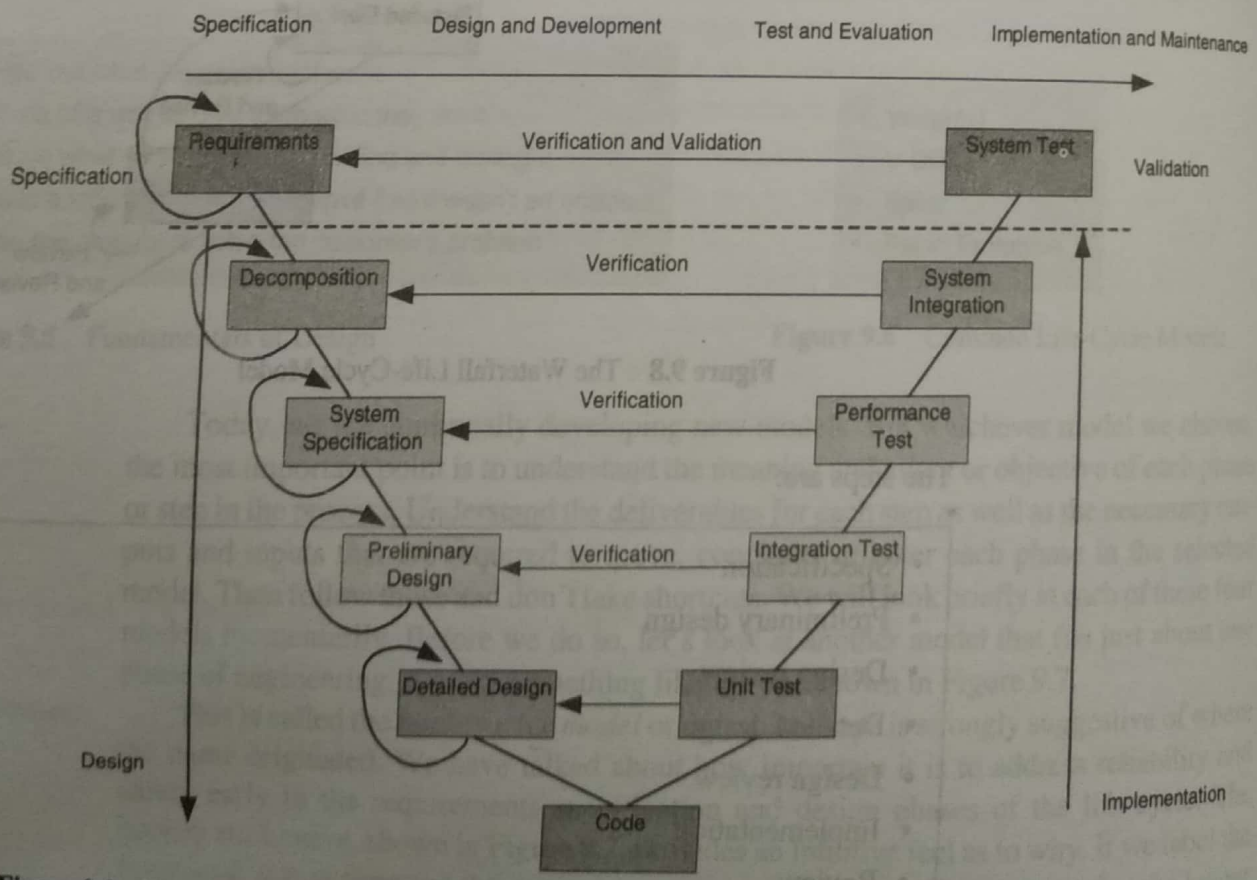


Figure 9.9 The V Life-Cycle Model

It is evident that each development activity builds a more detailed model of the system and that each verification step tests a more complete implementation of the system against the requirements at that phase. The development concludes the design and design-related test portion of the development cycle of the system with both a verification and a validation test against the original specification.

9.2.3 The Spiral Model

Spiral model, A Spiral Model of Software Development and Enhancement

The *Spiral model* was proposed and developed by Barry Boehm in *A Spiral Model of Software Development and Enhancement* (Computer, May 1988). A simplified version of that model is presented in Figure 9.10,

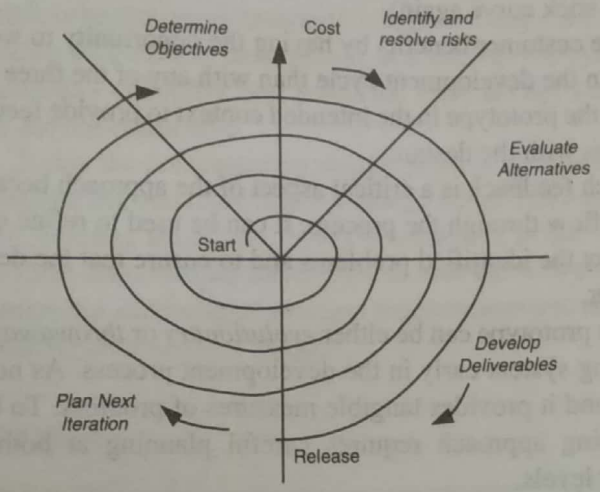


Figure 9.10 The Spiral Life-Cycle Model

The model takes a risk-oriented view of the development life cycle. Each spiral addresses the major risks that have been identified. After all the risks have been addressed, the Spiral model terminates, as did the Waterfall and V models, in the release of a product.

Like the earlier models, the Spiral model begins with good specification of the requirements. It then iteratively completes a little of each phase. Its philosophy is to start small, explore the risks, develop a plan to deal with the risks, and commit to an approach for the next iteration. The cycle continues until the product is complete. Boehm's model contains a lot more detail than the one presented in Figure 9.11. In both cases, each iteration of the spiral involves six steps.

- Determine objectives, alternatives, and constraints.
- Identify and resolve risks.
- Evaluate alternatives.
- Develop deliverables—verify that they are correct.
- Plan the next iteration.
- Commit to an approach for the next iteration.

Figure 9.11 Spiral Life-Cycle Model Steps

The Spiral model is an improvement on the Waterfall and V models because it provides for multiple builds as well as several opportunities for risk assessment and customer involvement. On the negative side, it is elaborate, difficult to manage, and does not keep all developers occupied during all of the phases.

9.2.4 Rapid Prototyping—Incremental

Rapid Prototyping model

The *Rapid Prototyping model* is intended to provide a rapid implementation (hence the name) of high-level portions of both the software and the hardware early in the project. The approach allows developers to construct working portions of the hardware and software in incremental stages. Each stage consists of design, code and unit test, integration test, and delivery. At each stage through the cycle, one incorporates a little more of the intended functionality.

The prototype is useful for both the designer and the customer. For the designer, it enables the early development of major pieces of the intended functionality of system. By doing so, it helps to establish and verify the structural architecture as well as the control flow through the system. Such an approach permits one to identify major problems early (the hockey stick curve again).

The customer benefits by having the opportunity to work with a functional unit much earlier in the development cycle than with any of the three previous models. The customer can use the prototype in the intended context to provide feedback to the designers about any problems with the design.

Such feedback is a critical aspect of the approach because it encourages backwards or reverse flow through the process. It can be used to refine or change the prototype in order to correct the identified problems and to ensure that the design meets the real needs of the customer.

evolutionary, throwaway

The prototype can be either *evolutionary* or *throwaway*. It has the advantage of having a working system early in the development process. As noted, problems can be identified earlier, and it provides tangible measures of progress. To be effective, however, the rapid prototyping approach requires careful planning at both the project management and designer levels.

Be careful how the prototype is used:

> ***Caution:*** The prototype should never turn into the final product.

Let's now move into the design process. Design begins with the real world where we are trying to solve problems in order to make our lives easier.

9.3 PROBLEM SOLVING—FIVE STEPS TO DESIGN

When we begin the design of a new product or have to incorporate several new features or capabilities into an existing one, we begin with a set of requirements usually stated in text form. The goal is to map those requirements—the real world—through a series of transformations into a solution—the abstract world. During the design process, we move from the concrete, real world into the abstract. These steps comprise what we describe as good design engineering practices.

Hopefully, we learned years ago that the first step to design is not to grab the nearest keyboard or processor and start hacking out code or wiring parts together. With today's complex systems, planning and thought before starting are essential to any successful design. If one takes the central elements from each of the life-cycle models, one finds that good system designers and successful projects generally proceed using a minimum of five steps (see Figure 9.12).

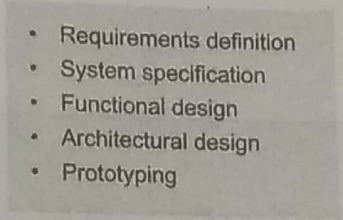


Figure 9.12 Five Steps to a Successful Design

The formality of each step depends on the complexity of the end product. If you are working alone or with several others in your own company on a smaller project, a white board in the center of the garage can often suffice. If you are orchestrating a project that includes developers, manufacturers, and regulations in several countries around the world (which is becoming increasingly common today), the need for formality increases. When working with each of these phases of a product life cycle, we must remember that they are guidelines—collective best practices. They are not a checklist to a successful project, and they are not exhaustive.

IP (intellectual property)

Today the contemporary design process must also enforce *IP (intellectual property)*, capitalization, and reuse at every design stage. The days of Bob Widler (the father of the op amp) lecturing about integrated circuit design in the bars of Silicon Valley are long gone. One must also consider traceability in both the forward and reverse directions. Traceability captures the relationships between requirements and all subsequent design data and helps in managing requirements changes.

.4 THE DESIGN PROCESS

As we begin to explore the product development cycle, we will walk through each of these five steps. Rather than focus on how one particular model approaches the interpretation of these steps, we will try to identify the essential elements of each step. The approach that we will present is top down and iterative.

The first two steps focus on capturing and formalizing the external behavior of the system. The remaining three move inside the system and repeat the process for the internal implementation that gives rise to the desired and specified behavior. As we will do from the outside and on the inside, we will move from the general to the specific, capturing and specifying each aspect of the design.

A major task, once we move inside the system, will be to decompose and refine the design from a nebulous entity that someone needs into the product that implements that need. We will first decompose (organize) the collection of customers' wishes into functional blocks, which are then mapped into an architecture. That architecture provides the aggregate of hardware and software modules that will make up the ultimate system. The final step in the design cycle is to bring the design together into a prototype and ultimately into production.

Because there is not one right answer, the problem represents a challenge and an opportunity to be creative. A colleague who worked on numerous designs of a particular piece of measurement technology once said, "although each design performs exactly the same function, each also represents an opportunity to explore a new approach that is better than the old." That colleague built a career around doing what everyone else, including some of the top names in the industry, said couldn't be done.

One of the best ways to learn how to do something is simply to do it. So, let's get started. As we walk through each of the steps in the design process that we have identified, we will see how they apply to the following design. We begin with a textual description.

EXAMPLE 9.0

Designing a Counter

Stating the Problem

As a senior development engineer at *Your Time Is Our Frequency, Ltd.com*, you've just finished one project and are now getting ready to head off to the next. As part of the early planning of that project, you and one of the marketing folks are traveling around the country talking with people from a number of different engineering firms. You are trying to determine what features your customers would like to see in the next generation product.

You've been on the road with this guy for a couple of weeks now and are anxious to get home. All the cities are beginning to look exactly alike. Tuesday, this must be Cleveland . . . hmmm, looks just like the last three cities. Oh well. This is the last customer for this trip. This morning, you're talking with *High Flying Avionics, Inc.* They're interested in a new counter that can be used on several of their avionics production lines.

Following several hours of discussion with one of the manufacturing managers, you identify most of their requirements. Your discussion with them follows.

Business is a little slow right now and money is tight, so we don't have a large budget to purchase a lot of different new instruments. In fact, ideally, we'd like to be able to use the same instrument on several of our lines.

Today, we have our technicians running most of the tests manually, but, in future, we'd like to be able to automate as many of these tests as we can. As we upgrade our systems, we'd like to be able to operate several of these counters remotely from a single PC. Here are some of the other things that we'd like to be able to do.

As part of our ongoing efforts to improve production and flow through our lines, we monitor the rate at which units arrive into each of the major assembly areas. To do that, we need to be able to track how many of our navigation radios come down a production line each hour. Because we support small-quantity builds of different kinds of radios, the rate at which the units come past the monitoring points is not constant. As each radio arrives at an entry point, it breaks an IR beam. On most of the lines, breaking the beam generates a 1-μsec-wide, negative going 5.0 V pulse. However, we do have several older lines that we must still support. On these, the pulse is positive going.

On several of the newer lines, we have to measure frequency up to 150,000 MHz. We also have several tests for which we must measure frequencies in the range of 50 KHz ± 0.001 KHz and 100 Hz with 0.001 Hz resolution. On another line, we have several instruments with output signals that have a duration up to 1.0000 ± 0.0001ms and others that have a duration of up to 9.999 to 10.000 ms and up to 1.000 ± 0.001sec. These signals are not periodic. Finally, we have several periodic signals on those same units that we must be able to measure with the same accuracy and resolution.

9.5 IDENTIFYING THE REQUIREMENTS

The development of a well-conceived and well-designed system must begin with a requirements definition. Such a need holds, independent of the life-cycle model that one chooses to work with. Unlike the people in the drawing in Figure 9.13 paraphrased from an unknown author, we cannot begin a design until we know what we are supposed to be designing.

The goal of the requirements identification process is to capture a formal description of the complete system from the customer's point of view and then to document these needs as written definitions and descriptions. Such documentation forms the subsequent basis for the formal design specification.

Very often, we use the natural language of the customer and of the application context. We do so because such a formal expression of the requirements forces the early discussion and resolution of many complex problems, involving a variety of people with expertise in many different areas, particularly those who are knowledgeable in the application domain. We express the role that the requirements definition plays between the customer and those who execute the design with the accompanying simple graphic in Figure 9.14.

Figure 9.13 Starting a New Project

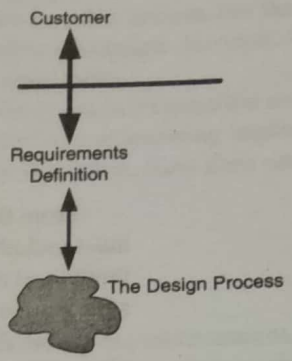


Figure 9.14 The Interface Between the Customer and the Design Process

The requirements definition provides the interface between the customer and the engineering process. It is the first step in transforming the customer's wishes into the final product. One can see, then, that the requirements definition is a description of something that is wanted or needed. It identifies and captures a set of required capabilities or operations. As one begins to identify all the requirements, it is important to consider *both* the system to be designed *and* the environment in which it is to operate.

At this early stage in the product life cycle, the goal is to capture and express purely external views of the environment, the system, and their interaction. With respect to the system, one refers to such a view as its public interface. One tries to identify *what* needs to be done (and *how well* it needs to be done) starting with the user's needs and requirements.

what
how well

The first view of the environment and of the system takes the form shown in Figure 9.15. It is evident that the environment surrounds the system. The inputs to and outputs from the system can come from or go to anywhere in the environment. As one begins, one should make no assumptions about the extent of either.

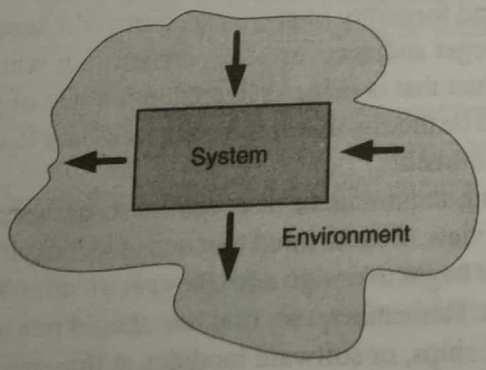


Figure 9.15 The System and Its Environment—Step 0

The first step is to abstract and consolidate that view so that both appear as the black boxes seen in Figure 9.16.

The initial focus must be on the world or environment (the application context) in which the system is to operate. Next, one follows with an increasingly detailed description of the role played by the system in that environment, and at each step one adds to and refines the requirements.

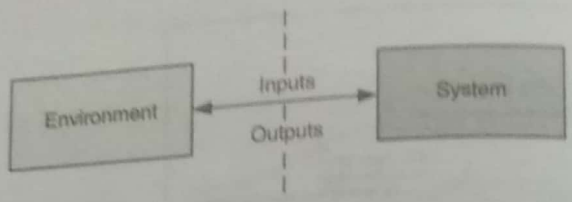


Figure 9.16 The System and Its Environment—Step 1

From the perspective of the environment, one can see that the requirements definition must include a specification for the containing environment, a description/definition of the inputs and outputs to and from that environment, a description of necessary behavior of the system, and a description of how the system is to be used.

From the system's point of view, one starts at a high level of abstraction with an *outside* view. One develops the definition(s) that are appropriate for that level. As was done when specifying the environment, through progressive refinement, one moves to lower levels of abstraction and a more detailed understanding and definition.

At this stage in the development life cycle, as the definition of the requirements solidifies and is ultimately formalized into a specification, one should be unencumbered by plans for implementation. The focus should be on the high-level behavior of the system. The complete, accurate, and internally consistent specification must be available before one can start formal design. Ideally, it should be executable and thereby able to work in conjunction with a modeling tool suite. Such an executable specification ultimately serves as the basis for validation of the system.

Although an executable specification is a laudable goal, achieving that goal can become difficult when one must include support for nonfunctional constraints, integrate legacy components into an abstract model, and potentially combine different domain-specific languages and semantics.

9.6 FORMULATING THE REQUIREMENTS SPECIFICATION

Let's now examine some of the things that one should think about when starting to identify and capture the requirements and when trying to define them in a formal specification. The form, extent, and formality of such a specification depends on the project on which one is working, the target audience, and the company for which one is working. Remember, too, that it is a product that is being delivered, not a pile of paper. As a rule of thumb, the specification should be the absolute minimum necessary to capture and clearly identify all of the necessary requirements.

requirements specification

In capturing requirements, one strives to be very specific about the details from the user's point of view. Bear in mind that one is identifying and formalizing the *requirements*. One still cannot begin to design until the *specification* has been completed and the customer has agreed to it. Remember, too, that one should not be discussing microprocessors, memory, peripheral chips, or software modules at this point in the development process.

As one begins the designs, one usually has some general ideas, casual discussion, and thoughts but nothing firm. One can use these as a guide in directing the steps, but one cannot design from them. It is important to be careful, however, not to rely too heavily on preconceived ideas; one should always be open to alternative approaches. Starting to code or draw logic diagrams at this point is inviting major problems as the project proceeds. In all likelihood, the project will fail.

For the environment, one must establish a description of all relevant entities and of the behaviors of all activities. One must know how the environment is interacting with the system and the effects on the environment by the outputs from the system. For the system, one requires a description of all inputs and outputs as well as a complete description of the functional and operational behaviors and technological constraints.

At this juncture we can naturally ask: how can one get such information about (let alone model) the system and the environment without describing or knowing implementation of the system? The internals are inherently unknown at this point. How does one capture the desired behaviors?

9.6.1 The Environment

A reasonable first step begins with defining and describing the environment, the world in which the system must operate. The environment is a temporal world; it is a heterogeneous collection of entities of one form or another. It comprises the collection of physical devices to which the system is interconnected as well as any physical world attributes that the system intends to measure or control or that can have an effect on the system. The initial goals in understanding the environment are to identify all relevant entities, then characterize their effects on the system, and vice versa. When the requirements specification has been completed, one should have all the necessary information about such entities, with sufficient detail to support solving the problem.

9.6.1.1 Characterizing External Entities

Each entity that makes up the environment is characterized by a name and an abstracted public interface. That interface consists of the entity's inputs and outputs as well as its functional behavior. The specification of the external environment should contain the following for each entity.

- *Name and Description of the Entity*
  The name should be suggestive of what the entity is or does. The description should present the nature of the entity. Is it data, an event, a state variable, a message? An entity may be something that is to be controlled—for example, the rudder on an aircraft or the clear air turbulence that must be accounted for in such a control system.
- *Responsibilities—Activities*
  What activities or actions is the environment expected to perform? The hydraulic system moving the rudder is part of the environment. Its action or responsibility is to move the rudder in response to the signal coming from the system being designed.
- *Relationships*
  What are the relationships between the entity and its responsibilities or activities? Is that relationship causal or responding? Is it a producer or a consumer?
- *Safety and Reliability*
  Safety and reliability issues must be included early in the specification process. With respect to the environment, at the requirements stage, the focus is primarily on safety. The goal is to identify all safety critical issues and hazards so that they can be addressed in detail in the system design specification. One should also identify any regulatory agencies under whose auspices the system will operate.

9.6.2 The System

The focus next shifts to the system's point of view. The same questions posed for the environment are now asked about the system. As with the characterization of the environment, the initial goals are to identify all the aspects of the public interface of the system and then characterize their effects on the environment and vice versa.

9.6.2.1 Characterizing the System

Characterization of the system begins with identifying inputs and outputs.

- *System Inputs and Outputs*
  The system interacts with the real world through the entities described and defined in the environmental characterization. The inputs to the system are the outputs from environmental entities, and the outputs from the system are the inputs to the environmental entities. One can easily see that the system I/O has already been characterized in environmental entity specification.

  For each such I/O variable, the following information is already available:

  - The name of the signal
  - The use of the signal as an input or output
  - The nature of the signal as an event, data, state variable, and so on

  Working with the environment specification, one can write the structure, domain of validity, and physical characteristics of each signal. To these, one can add any technical or technological constraints that are identified.

- *Responsibilities—Activities*
  As was done with the specification of the environment, focus now turns to the function that the system is intended to perform. Before it is designed, the system appears as a black box. It can only be viewed from an external point of view. A section on functional behavior is now included in the specification.

  The functional description defines the external behavior of the system. It characterizes the effects of the system outputs on the environmental entities and the system's intended response to inputs from the environmental entities. It elaborates on how the system is used and to be used by the user. Such a specification is equivalent to developing a model of the system.

  The functional description can be captured in a variety of ways. One effective approach is to use the UML tools discussed earlier. One can construct one such view through use case and class diagrams. Another view can be gained through high-level state charts and activity diagrams; data and control flow diagrams commonly used in structured design methodologies give a third view.

  As one formulates these diagrams and the specification, care must be taken to ensure that the specified (and ultimately modeled) states are appropriate to the application. One must make certain that the actions that are captured in the specification accurately reflect the desired (external) behavior of the system as perceived and intended by the customer. In the specification, one must ensure that the conditions or constraints on its behavior are only a function of the inputs coming into the system, the specified states, the internal events, and the appropriate time demarcation (relative or absolute).

- *Safety and Reliability*
  In formulating the safety and reliability requirements for the system, the focus is on the high-level objectives of each and on the strategy for achieving those goals.
  The safety considerations should address
  - Safety guidelines, rules, or regulations under the governing agencies identified under the environment portion of the specification.

  With respect to reliability, one can specify
  - The system uptime goals
  - Potential risks, failures, and failure modes
  - Failure management strategy

EXAMPLE 9.0
esigning a Counter
Cont.)

Identifying the Requirements

Starting from the trip report from *High Flying Avionics, Inc.*, which discussed their needs for a new counter, let's put the requirements specification together.

As a first step in the thought process, one extracts and summarizes the essential information from the trip report. By doing so, one can begin to focus on what should be included in the requirements specification. From the discussions with the customer, a high-level sketch of the system and the environment captures the essential parts of the problem. The next step is to begin to formalize the model of the system and the environment as illustrated in Figure 9.17.

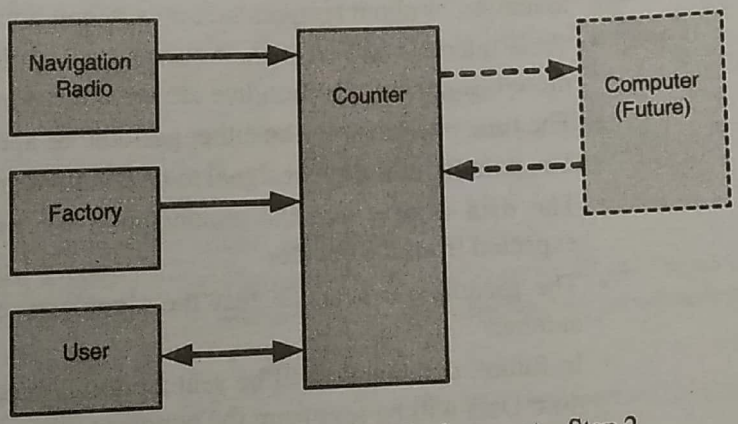


Figure 9.17 The System and Its Environment—Step 2

In its initial configuration, the environment contains:

- A set of navigation radios that are to be tested
- The user who is doing the testing
- The factory

Signals flow from the navigation radio to the counter, but not the reverse. The factory has inputs to the counter as well; these include the power system and the ambient environment in the factory. The user's interaction is bidirectional. The user must select and configure the measurement to be made and then view the results once the measurement is complete. For the computer, the signal interchange with the counter similarly occurs in both directions.

In the developing model, the factory can be viewed as an aggregation of test lines and the radios to be tested. Later, the remote computer is to be added. The system to be designed, that is, the counter, interacts with all three entities. Such an interaction is reflected in Figure 9.18.

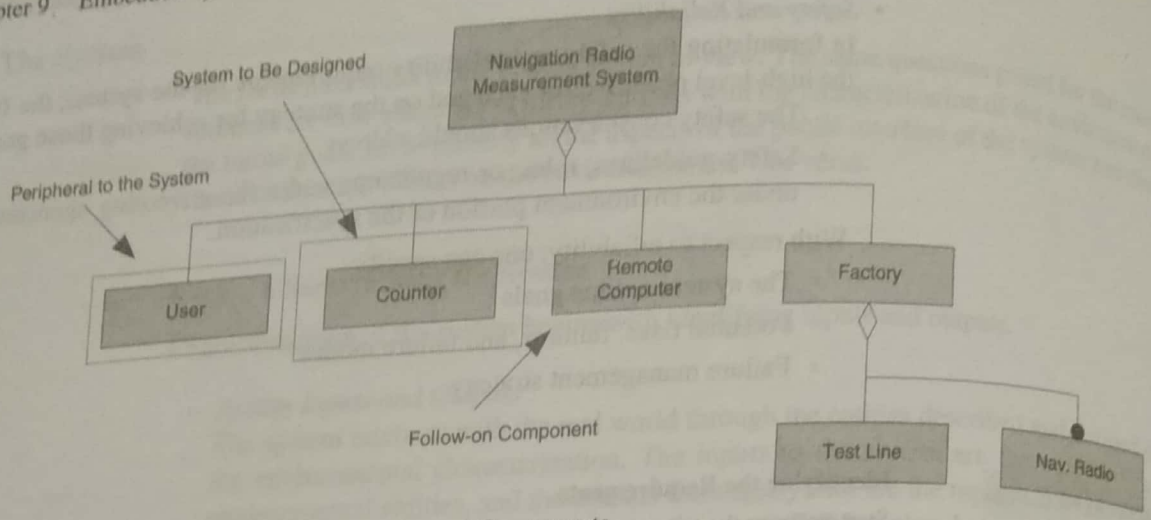


Figure 9.18 The System as an Aggregation of Components

Now let's move to the next level of detail.

ENVIRONMENT

- The customer has stated that the counter is to operate in a factory environment on any of several productions lines. Based on such an understanding, one can make certain assumptions about temperature, power, and ambient lighting.
- Time intervals and frequencies on the navigation radios and events from equipment monitoring the production line are to be measured.
- The time intervals may be either periodic or aperiodic but cannot be both.
- The polarity of the event signal to be counted can be either positive or negative going.
- The data display and the annunciation for mode and range are the only outputs expected from the counter.
- The assumption is made that the signals to be measured are independent of one another.
- In future, commands will be sent from a computer to the counter to direct its operation. Data will be sent from the counter to the computer.

COUNTER

- The counter must have the ability to measure time intervals and frequencies and to count events.
- The frequencies are fixed but span a range of values.
- The time intervals span a range of values and may be either periodic or aperiodic, but they cannot be both.
- The counter will support the user's ability to manually select mode and measurement range for all input signals.
- The counter will continue to make and display the selected attribute of the signal until power to the system is turned off or until the user makes another selection.
- The counter will measure only one signal at a time.
- An event can be modeled as an aperiodic time signal.
- The design will be sufficiently flexible to allow future inclusion of the ability to send commands from a computer to the counter to direct its operation.

- The response of the counter to remote commands will be the same as its response to front panel selections, with the exception that measured data will be sent from the counter to the computer as well as to the front panel display.

System Requirements Specification

The next step is to formalize, in a specification, what is known about the system to be designed. The document, the *System Requirements Specification*, opens with a summary of the design.

System Requirements Specification for a Digital Counter

System Description

This specification describes and defines the basic requirements for a digital counter. The counter is to be able to measure frequency, period, time interval, and events. The system supports three measurement ranges for each signal and two for events. The counter is to be manually operated with the ability to support remote operation in future. The counter is to be low cost and flexible, so that it may be utilized in a variety of applications.

Specification of External Environment

The counter is to operate in an industrial environment in a commercial grade temperature and lighting environment. The unit will support either line power or battery operation.

System Input and Output Specification

System Inputs

The system shall be able to measure the following signals.

Frequency in three ranges

- High range up to 150.000 MHz
- Midrange up to 50.000 KHz
- Low range up to 100.000 Hz

Period in three ranges

- High resolution up to 1.0000 ms
- Midresolution up to 10.000 ms
- Low resolution up to 1.000 sec

Time interval in three ranges

- High resolution up to 1.0000 ms
- Midresolution up to 10.00 ms
- Low resolution up to 1.000 sec

Events—up to 99 events in 1 minute

All signal inputs will be

- Digital data
- Voltage range 0.0 to 4.5 VDC

System Outputs

The system shall measure and display the following signals using a 6-digit display.

Frequency in three ranges

- High range up to 200.000 ± 0.001 MHz
- Midrange up to 200.000 ± 0.001 KHz
- Low range up to s200.000 ± 0.001 Hz

Period in three ranges

- High resolution up to 2.000 ± 0.0001 ms
- Midresolution up to 20.00 ± 0.01 ms
- Low resolution up to 2.000 ± 0.001 sec

Time interval in three ranges

- High resolution up to 2.0000 ± 0.0001 ms
- Midresolution up to 20.00 ± 0.01 ms
- Low resolution up to 2.000 ± 0.001 sec

Events in two ranges

- Fast up to 200 events in 1 minute
- Slow up to 2,000 events in 1 hour

User Interface

The user shall be able to select the following using buttons and switches on the front panel of the instrument.

Mode
  Frequency, Period, Time Interval, Events
  Range
    Frequency, Period, Time Interval—High, Mid, Low
    Events—Fast, Slow

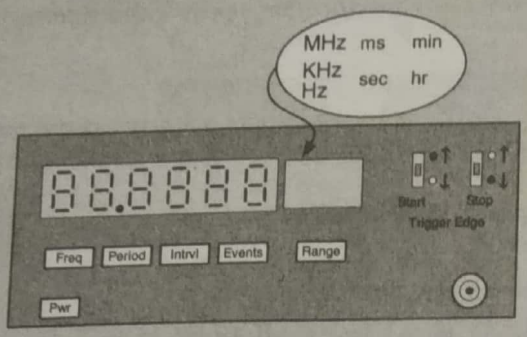
Trigger Edge
  Frequency, Period, and Events
    Rising or falling edge
  Time Interval
    Rising to rising edge
    Falling to falling edge
    Rising to falling edge
    Falling to rising edge

Reset

Power ON/OFF

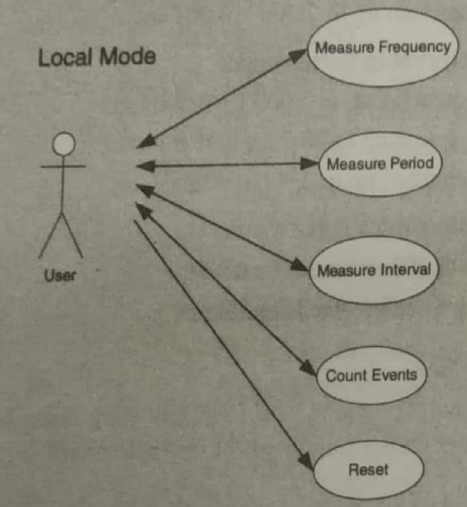
The measurement results shall be presented on a 6-digit display; leading zeros will be suppressed. The display shall be readable in direct sunlight and from any angle.

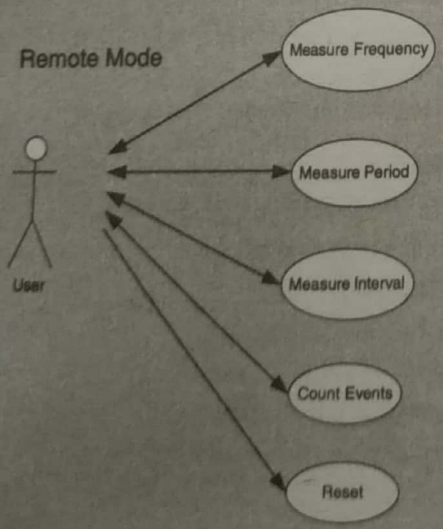
The front panel will appear as follows.



Use Cases

The use cases for the counter are given in the following two diagrams.





The first indicates manual operation through the front panel, and the second through a remote connection to a computer.

The remote option will not be included in the initial model, but will be incorporated in a later release. The time of that release is to be determined.

Execution of the selected measurement function will not depend on how (local or remote) that function was selected.

At power ON, the default mode is to measure frequency. All ranges will default to their highest value.

Measure Frequency The counter will continuously measure and display the frequency of the input signal on the currently selected range as long as the *Frequency* mode is selected.

If the frequency of the input signal exceeds the maximum allowable value on the selected range, the display will present the full-scale reading and will flash.

If the frequency of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading.

If the input signal returns to a value within the bounds of the range, the value of the frequency will be displayed.

The range may be changed at any time by depressing the *range select* pushbutton.

The user may elect to measure frequency starting on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

Measure Period The counter will continuously measure and display the period of the input signal on the currently selected range as long as the *Period* mode is selected.

If the period of the input signal exceeds the maximum allowable value on the selected range, the display will present the full-scale reading and will flash. If the period of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading.

If the input signal returns to a value within the bounds of the range, the value of the period will be displayed.

The range may be changed at any time by depressing the *range select* pushbutton.

The user may elect to measure period starting on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

Measure Interval The counter will continuously measure and display the duration of the selected portion of the input signal on the currently selected range as long as the *Interval* mode is selected.

If the duration of the selected portion of the input signal exceeds the maximum allowable value on the selected range, the display will present the full-scale reading and will flash.

If the duration of the selected portion of the input signal is below the minimum allowable value on the selected range, the display will display zero.

If the input signal returns to a value within the bounds of the range, the value of the duration of the selected portion of the input signal will be displayed.

The range may be changed at any time by depressing the *range select* pushbutton.

The user may elect to commence measuring the interval on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

The user may elect to terminate the measurement interval on the positive or negative edge of the signal by depressing the *stop trigger edge* pushbutton.

Note that the signal duration from positive edge to positive edge or negative edge to negative edge is the same as the period of the signal.

Events The counter will continuously count and display the number of occurrences of the input signal on the currently selected range. The accumulated count will be reset to 0 at the end of the select count duration.

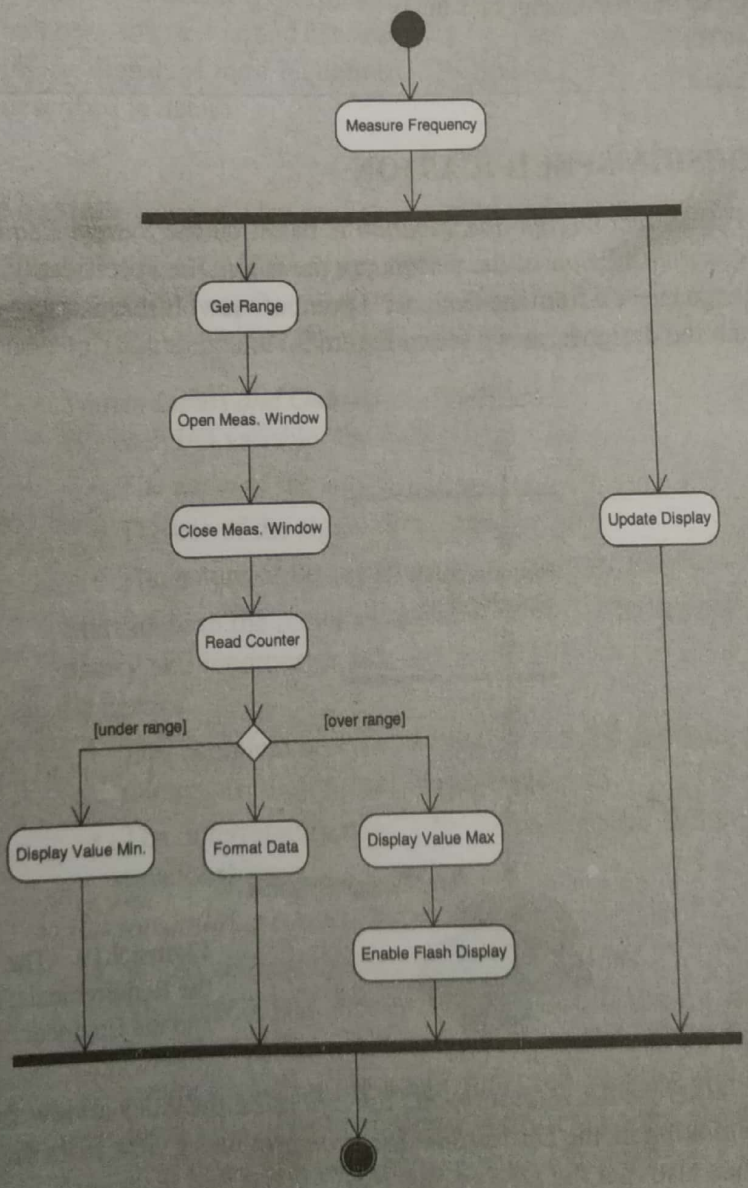
The range may be changed at any time by depressing the *range select* pushbutton.

The user may elect to increment the count on the positive or negative edge of the input signal by depressing the *start trigger edge* pushbutton.

If the number of accrued counts exceeds the maximum allowable value on the selected range, the display will present the full-scale reading and will flash.

System Functional Specification

The system is intended to make four different kinds of digital measurement in the time and frequency domains comprising frequency, period, time interval, and events. The activities associated with the *measure frequency* mode are shown in the following diagram.



The time and frequency measurements will be implemented to provide three user selectable resolution ranges: high frequency range/shorter duration signals, a second for midrange frequency/midrange duration signals, and a third for low frequency/longer duration signals. The events measurement capability will support two selectable counting durations, shorter and longer.

For frequency, period, and events measurements, the user will be able to select either a positive or negative edge trigger. For interval measurements, the user will be able to select the polarity of the start and stop signals independently.

Operating Specifications

The system shall operate in a standard commercial / industrial environment

Temperature Range 0–85C

Humidity up to 90% RH noncondensing

Power 120–240 VAC 50 Hz, 60 Hz, 400 Hz, 15 VDC

The system shall operate for a minimum of 8 hours on a fully charged battery

The system time base shall meet the following specifications.

Temperature stability 0–50 C

$< 6 \times 10^{-6}$

Aging Rate

90 day $< 3 \times 10^{-8}$

6 month $< 6 \times 10^{-7}$

1 year $< 25 \times 10^{-6}$

Reliability and Safety Specification

The counter shall comply with the appropriate standards

Safety: UL-3111-1, IEC-1010, CSA 1010.1

EMC: CISPR-11, IEC 801-2, -3, -4, EN50082-1

MTBF: Minimum of 10,000 hours

9.7 THE SYSTEM DESIGN SPECIFICATION

System Design Specification, System Requirements Specification

The *System Design Specification* is based on the *System Requirements Specification* and specifies the *how* of the design, not the *what*. The specification is written in the designer's language and from the designer's point of view. It serves as a bridge between the customer and the designer, as we see in Figure 9.19.

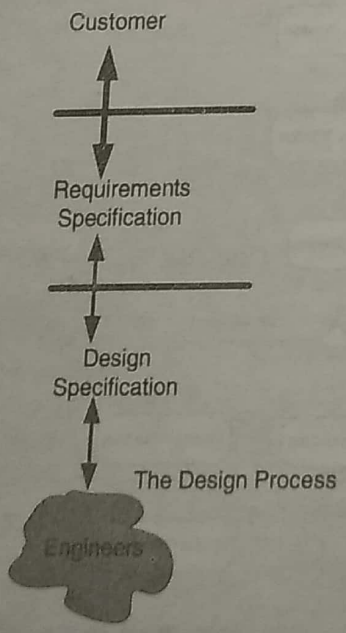


Figure 9.19 The Customer, the Requirements, the Design, and the Engineer

Requirements Specification Design Specification

Whereas the *Requirements Specification* provides a view from the outside of the system looking in, the *Design Specification* provides a view from the inside looking out as well. Notice also that the *Design Specification* has two masters:

- It must specify the system's public interface from inside the system.
- It must specify *how* the requirements defined for and by the public interface are to be met by the internal functions of the system.

Requirements Specification
Design Specification

We have seen that the *Requirements Specification* is written in less formal terms with the intent of capturing the customer's view of the product. The *Design Specification* must formalize those requirements in precise, unambiguous language. Putting the inevitable changes that occur during the lifetime of any project aside for the moment, we find that the design specification should be sufficiently clear, robust, and complete that a group of engineers could develop the product without ever talking to the author of the specification.

Design Note

A good litmus test of the viability of a design specification is the question, "If I send this to my colleague (who is working for one of our subcontractors), will he or she understand this?" If the answer is no, the specification should be reexamined.

9.7.1 The System

As part of formalizing and quantifying the system's requirements, one must attach concrete numbers, tolerances, and constraints to all of the system's input and output signals. All timing relationships must be defined. The system's functional and operational behaviors are described in detail.

9.7.2 Quantifying the System

The quantification of the system's characteristics begins with the inputs and outputs, based on the specified requirements. The necessary technical details are added to enable the engineer to accurately and faithfully execute the actual design.

- *System Inputs and Outputs*
  For each I/O variable, the following are specified.
  - The name of the signal
  - The use of the signal as an input or output
  - The nature of the signal as an event, data, state variable, and so on.

  Starting with the requirements specification, we provide detailed descriptions as necessary and incorporate any additional technical or technological constraints that may be needed.
  - The complete specification of the signal, including nominal value, range, level tolerances, timing, and timing tolerances
  - The interrelationships with other signals, including any constraints on those relationships
- *Responsibilities—Activities*
  - Functional and Operational Specifications
    The functional and operational specifications that will quantify the dynamic behavior of the system are now formulated. The functional requirements specification identifies the major functions that the system must perform from a high-level view. The operational specification endeavors to capture specific details of how those functions behave within the context of the operating environment.

The manner in which a particular function must operate, the conditions imposed on the operation, and the range of that operation are now captured. The specification must consider concrete numbers—precisions and tolerances.

All variables in the functional specification, all operating conditions, and all ordinary and extraordinary operating modes must be quantified. The specification may include domain-specific knowledge that is proprietary or heuristically known to the customer. Such knowledge can be very important to the design.

In stating the specific design requirements for the system, one can use tables, equations or algorithms, formal design language, or pseudo code, flow diagrams, or detailed UML diagrams such as state charts, sequence diagrams, and time lines. Schematics, codes, or parts lists are not included, except in limited circumstances.

- Technological (and Other) Specifications
  The technological portion includes all detailed and concrete specifications that are relevant to the design of the system hardware and software. Five areas that should be considered can easily be identified.

1. Geographical constraints
   Distributed applications can span a single room, can expand to include a complete factory, or can encompass several countries. Consequently, one must address both the technical items such as interconnection topologies, communications methods, restrictions on usage, and environmental contamination as well as nontechnical matters such as costs associated with the physical medium and its installation.
2. Characterization of and constraints on interface signals
   The assumption is made that signals between the system and the external world are electrical, optical, or wireless or that they can be converted into or from such a form. The necessary physical characterization of each is obviously going to depend on the type of signal. That is, an electrical signal is specified differently from an optical signal.

   Since many of the interface signals may be driven by the external environment, potentially they are beyond the designer's control. Therefore, it is important to gain as much information about them as possible.
3. User interface requirements
   If the system interfaces to such external world devices as medical or instrumentation equipment, how information is presented and whether any relevant and associated protocols exist must be considered. There may also be standards that govern how such information must be presented.

   Consider the significant risk that would arise if each avionics vendor presented critical flight information and controls to the aircraft pilot in a different way. The near disaster at Three Mile Island in 1979 arose, in part, because of the confusion caused by too much information.
4. Temporal constraints
   The system may have to perform under hard or soft real-time constraints. Such constraints may specify delays on signals originating from external entities, responses to system outputs by external entities, and/or internal system delays.
5. Electrical Infrastructure considerations
   There must be a specification for the electrical characteristics of any electrical infrastructure. Included in this portion of the specification are power consump-

tion, necessary power supplies, tolerances and capacities of such supplies, tolerance to degraded power, and power management schemes.

- *Safety and Reliability*
  In formulating the design requirements for the safety and reliability of the system, the focus shifts to the detailed objectives of each and to the strategy for achieving those goals.

  Safety considerations should address
  - Understanding and specifying any environmental and safety issues

  The reliability specification should include
  - Requirements for diagnostic tests, remote maintenance, remote upgrade, and their details
  - Concrete numbers for MTTF and MTBF of any built-in self-test circuitry
  - Concrete numbers for MTTF and MTBF of the system itself
  - Consideration of system performance under partial or full failure

Let's now bring everything together.

EXAMPLE 9.0

Designing a Counter (Cont.)

Design Specification

Requirements Specification

Quantifying the specification

We will now continue with the development of the counter. The system *Design Specification* will follow, but extend, what has been captured in the *Requirements Specification*. The focus will now be on providing specific numbers, ranges, and tolerances for signals that are within the system.

Once again, we will put together any thoughts about the environment and the system prior to writing the specification.

Environment

Specifications relating to the environment have been discussed earlier. There are no changes here.

Counter

- When specifying measurement and stimulus equipment, the specifications for that equipment are generally 10 times (one order of magnitude) better than those for the signals that must be measured or generated.
- That margin is provided when specifying the range and tolerances on the counter's measurement capabilities.
- Specifications on counting events are based on the granularity of the timing of the interval during which the events are counted.
- The values to be displayed at the measurement boundaries are now defined.

The next step is to provide any additional detail that may be needed and to fully quantify the counter specifications.

System Design Specification for a Digital Counter

System Description

This specification describes and defines the basic requirements for a digital counter. The counter is to be able to measure frequency, period, time interval, and events. The system supports three measurement ranges for each signal and two for events. The counter is to be manually operated with the ability to support remote operation in future. The counter is to be low cost and flexible so that it may be utilized in a variety of applications.

Specification of External Environment

The counter is to operate in an industrial environment in a commercial grade temperature and lighting environment. The unit will support either line power or battery operation. Specific details are included under Operating Specifications.

System Input and Output Specification

System Inputs

The system shall be able to measure the following signals

Frequency in three ranges

- High range up to 150.000 MHz
- Midrange up to 50.000 KHz
- Low range up to 100.000 Hz

Period in three ranges

- High resolution up to 1.0000 ms
- Midresolution up to 10.000 ms
- Low resolution up to 1.000 sec

Time interval in three ranges

- High resolution up to 1.0000 ms
- Midresolution up to 10.00 ms
- Low resolution up to 1.000 sec

Events

- Events to 99 per minute
- Signal level 0–4.0 V ± 0.5 V
- Transition time $10ns \leq t_{rise} t_{fall} \leq 50ns$

Voltage Sensitivity

- 50 mV RMS to ± 5.0 V ac signal + dc signal

All signal inputs will be

- Digital data
- Voltage range 0.0 to 4.5 VDC

System Outputs

The system shall measure and display the following signals using a 6-digit display

Frequency in three ranges

- High range
  - Measure: 0 – 200 ± 0.0001 MHz
  - Display: 0 – 200.000 MHz
- Midrange up to 200.000 KHz
  - Measure: 0 – 200 ± 0.0001 KHz
  - Display: 0 – 200.000 KHz
- Low range up to 200.000 Hz
  - Measure: 0 – 200 ± 0.0001 Hz
  - Display: 0 – 200.000 Hz

Period in three ranges

- High resolution up to 2.0000 ms
  - Measure: 0 – 2.00000 ± 0.00001 ms
  - Display: 0 – 2.0000± 0.0001 ms
- Midresolution up to 20.00 ms
  - Measure: 0 – 20.0000 ± 0.0001 ms
  - Display: 0 – 20.000± 0.001 ms
- Low resolution up to 2.000 sec
  - Measure: 0 – 2.0000 ± 0.0001 sec
  - Display: 0 – 2.000 ± 0.001 sec

Time interval in three ranges

- High resolution up to 2.0000 ms
  - Measure: 0 – 2.00000 ± 0.00001 ms
  - Display: 0 – 2.0000± 0.0001 ms
- Mid resolution up to 20.00 ms
  - Measure: 0 – 20.0000 ± 0.0001 ms
  - Display: 0 – 20.000± 0.001 ms
- Low resolution up to 2.000 sec
  - Measure: 0 – 2.0000 ± 0.0001 sec
  - Display: 0 – 2.000 ± 0.001 sec

Events in two ranges

- Fast up to 200 events in 1 minute
  - Measure: 0 – 200 ± 1 event
  - Display: 0 – 200 ± 1 event
- Slow up to 2000 events in 1 hour
  - Measure: 0 – 2000 ± 1 event
  - Display: 0 – 2000 ± 1 event

User Interface

The user shall be able to select the following using buttons and switches on the front panel of the instrument.

Mode

Frequency, Period, Time Interval, Events

Range

Frequency, Period, Time Interval—High, Mid, Low

Events—Fast, Slow

Trigger Edge

Frequency, Period, and Events

Rising or falling edge

Time Interval

Rising to rising edge

Falling to falling edge

Rising to falling edge

Falling to rising edge

Reset

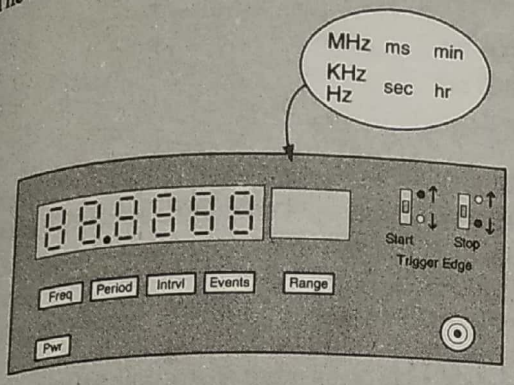
The reset button will clear the display to all 0's and reset the internal timing/counting chain.

The counter will be placed in the *frequency* mode with the *range* set to KHz, and the *trigger edge* set to *rising*.

Power ON/OFF

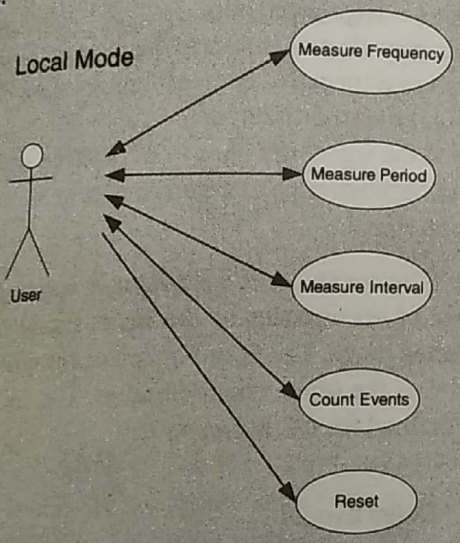
The measurement results shall be presented on a 6-digit LED display; leading zeros will be suppressed.

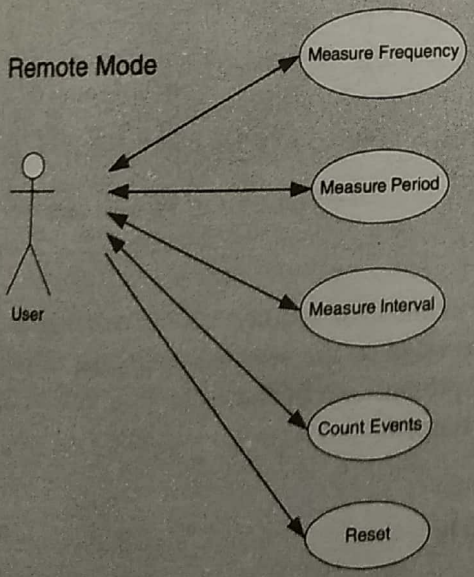
The decimal point will move to reflect the proper value for the range selected as the range pushbutton is pressed. The front panel will appear as follows.



Use Cases

The use cases for the counter are given in the following two diagrams.





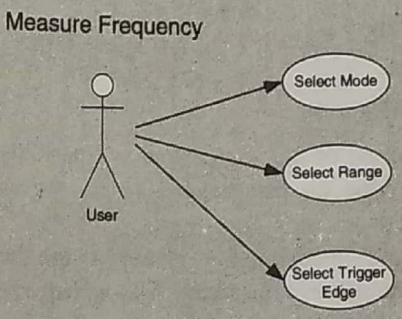
The first indicates manual operation through the front panel, and the second through a remote connection to a computer.

The remote option will not be included in the initial model but will be incorporated in a later release. The time of that release is to be determined.

Execution of the selected measurement function will not depend on how (local or remote) that function was selected.

At power ON, the default mode is to measure frequency. All ranges will default to their highest value.

Measure Frequency The counter will continuously measure and display the frequency of the input signal on the currently selected range as long as the *Frequency* mode is selected. The following use cases are defined for the *Frequency* mode.



If the frequency of the input signal exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based on the selected range,

- 200.000 MHz
- 200.000 KHz
- 200.000 Hz

If the frequency of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading.

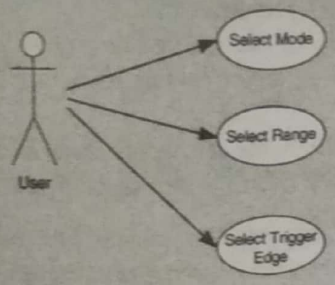
If the input signal returns to a value within the bounds of the range, the value of the frequency will be displayed.

The range may be changed at any time by depressing the *range select* pushbutton.

The user may elect to measure frequency starting on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

Measure Period The counter will continuously measure and display the period of the input signal on the currently selected range as long as the *Period* mode is selected. The following use cases are defined for the *Period* mode.

Measure Period



If the period of the input signal exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based on the selected range:

- 2.0000 ms
- 20.000 ms
- 2.000 sec

If the period of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading.

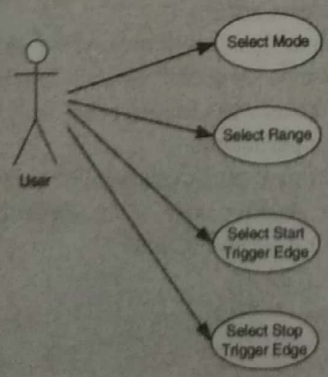
If the input signal returns to a value within the bounds of the range, the value of the period will be displayed.

The range may be changed at any time by depressing the *range select* pushbutton.

The user may elect to measure period starting on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

Measure Interval The counter will continuously measure and display the duration of the selected portion of the input signal on the currently selected range as long as the *Interval* mode is selected. The following use cases are defined for the *Interval* mode.

Measure Interval



If the duration of the selected portion of the input signal exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based on the selected range.

- 2.0000 ms
- 20.000 ms
- 2.000 sec

If the duration of the selected portion of the input signal is below the minimum allowable value on the selected range, the display will display zero.

If the input signal returns to a value within the bounds of the range, the value of the duration of the selected portion of the input signal will be displayed.

The range may be changed at any time by depressing the *range select* pushbutton.

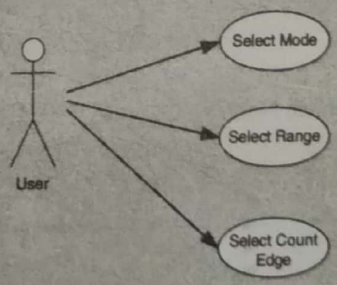
The user may elect to commence measuring the interval on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

The user may elect to terminate the measurement interval on the positive or negative edge of the signal by depressing the *stop trigger edge* pushbutton.

Note that the signal duration from positive edge to positive edge or negative edge to negative edge is the same as the period of the signal.

Events The counter will continuously count and display the number of occurrences of the input signal on the currently selected range. The accumulated count will be reset to 0 at the end of the select count duration. The following use cases are defined for the *Events* mode.

Count Events



If the number of accrued counts exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based on the selected range,

- 200 min
- 2,000 hr

The range may be changed at any time by depressing the *range select* pushbutton.

The user may elect to increment the count on the positive or negative edge of the input signal by depressing the *start trigger edge* pushbutton.

System Functional Specification

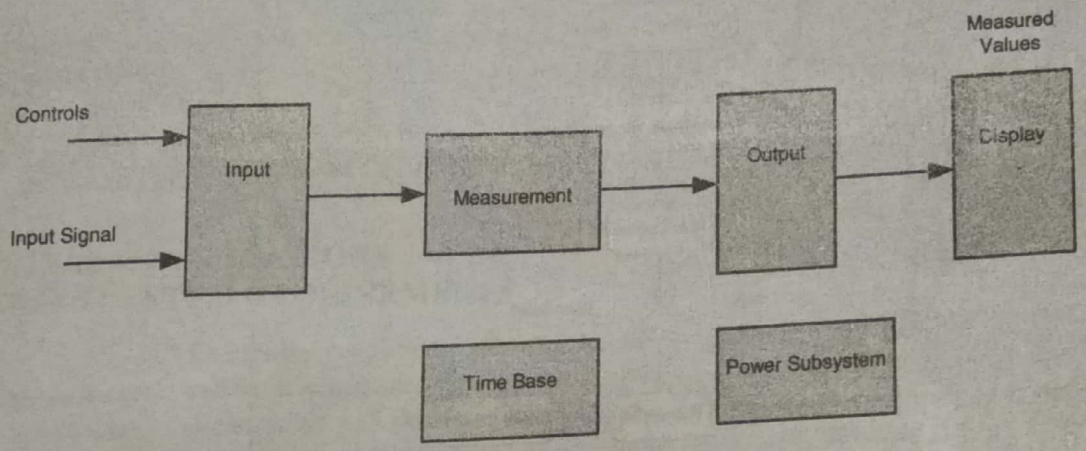
The system is intended to make four different kinds of digital measurements comprising frequency, period, time interval, and events.

The time and frequency measurements will be implemented to provide three user selectable resolution ranges: high frequency range/shorter duration signals, a second for midrange frequency/midrange duration signals, and a third for low frequency/longer duration signals. The events measurement capability will support two selectable counting durations, shorter and longer.

For frequency, period, and events measurements, the user will be able to select either a positive or negative edge trigger. For interval measurements, the user will be able to select the polarity of the start and stop signals independently.

The system will be designed so as not to preclude the incorporation of a remote access option in future.

The system comprises six major blocks as given in the following block diagram



Input Subsystem The input subsystem shall provide the ability for the user to select any of the measurement functions, ranges, and triggering polarities. The subsystem also selects and routes the input signal to the appropriate portion of the measurement subsystem.

Output Subsystem The output subsystem implements the range, edge selection, control information, and data formatting for proper presentation on the front panel display.

Time Base The time-base subsystem is a phase locked loop and divider chain driven from a 100-MHz crystal oscillator. This subsystem will provide two clock phases to drive the internal control and decision logic. Each phase will be 200.0000 ± 0.0001 MHz.

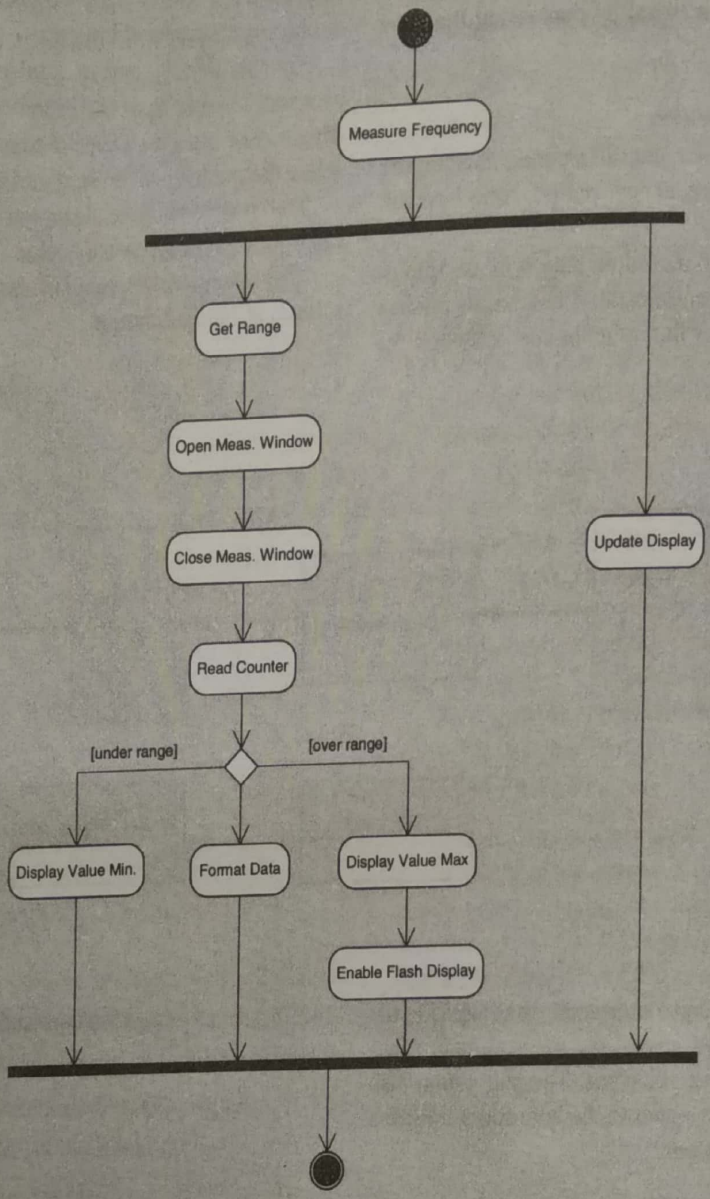
The time base will also provide the following frequencies that are used to define the measurement windows for the events and frequency measurements and provide the counting frequencies for the time interval and period measurements.

- Frequency—200.0000 ± 0.0001 MHz
- Period—100.0000 ± 0.0001 MHz
- Time Interval—100.0000 ± 0.0001 MHz
- Events—10.00 ± 0.01Hz

Measurement Subsystem The measurement subsystem provides the logic and control to execute the measurements of time and frequency.

- The frequency measurement will be implemented by opening a window for 1.00 ± 0.01 seconds. During the time the window is open, the measurement subsystem will gate the unknown input frequency into a 7 stage binary coded decimal (BCD) counter. When the window closes, the counter will contain the value of the unknown frequency.

The activities necessary to execute a frequency measurement are given in the following diagram.



- The period and time interval measurements will be made by opening a window on the specified signal edge. While the window is open, a frequency of 100.0000 ± 0.0001 MHz will be gated into a 7 stage BCD counter. When the window closes, the counter will contain the values of the unknown time interval.
- The counter will contain the number of events that occurred during the measurement interval.
- The events measurement will be made by opening a window for 1.00 ± 0.01 seconds for the fast mode and 3600.0 ± 0.1 seconds for the slow mode. During the time the window is open, the measurement subsystem will gate the unknown input to a 4 stage BCD counter. When the window closes, the counter will contain a measure of the number of events that occurred during the time interval.

Power Supply Subsystem The power supply subsystem will provide the following voltages at the specified current levels to the internal logic.

```
 +5.0 ± 0.01 VDC @ 10 A
+15.0 ± 0.01 VDC @ 500 mA
-15.0 ± 0.01 VDC @ 500 mA
```

At power on, there shall be a negative going reset signal. That signal shall remain in the low state for a minimum of 10 ms and shall have the ability to sink up to 1A.

Display The instrument display shall display the results of the selected measurement on a 6-digit, 7-segment red LED display. The layout of the major features and functions is given in the earlier diagram.

Operating Specifications

The system shall operate in a standard commercial/industrial environment.

Temperature Range 0–85C

Humidity up to 90% RH noncondensing

Power Automatic line voltage selection

- 100–120 VAC ± 10% 50, 60, 400 Hz ± 10%
- 220–240 VAC ± 10% 50, 60 Hz ± 10%

The system shall operate for a minimum of 8 hours on a fully charged battery.

Net weight/size 2.75 kg, H: 90 mm x W: 200mm x D: 300 mm

The system time base shall meet the following specifications.

Temperature stability 0–50 C

$< 6 \times 10^{-6}$

Aging Rate

90 day	$< 3 \times 10^{-8}$
6 month	$< 6 \times 10^{-7}$
1 year	$< 25 \times 10^{-6}$

Reliability and Safety Specification

The counter shall comply with the appropriate standards

Safety: UL-3111-1, IEC-1010, CSA 1010.1

EMC: CISPR-11, IEC 801-2, -3, -4, EN50082-1

MTBF: Minimum of 10,000 hours

9.8 SYSTEM SPECIFICATIONS VERSUS SYSTEM REQUIREMENTS

System Design Specification, System Requirements Specification

Examining the different steps that have been outlined up to this point, we find a lot of duplication. It would seem that the *System Design Specification* and *System Requirements Specification* are just different names for the same thing. But they are not; requirements and specifications are fundamentally different types of descriptions.

> ***Requirements*** Give a description of something wanted or needed. They are a set of needed properties.

what
how well, how, System Design Specification

Generally, requirements come from the marketing or sales department, and they represent the customer's needs. The requirements definition and specification is not concerned with the internal organization of the system. Rather, it is intended to describe *what* a system must do and *how well* it has to do it, not *how* it does it. The *System Design Specification* is generated by engineering as an answer to and a description of how to implement the requirements. Then the two groups negotiate and iterate until the requirements and specifications are consistent.

> ***Specification*** is a description of some entity that has or implements those properties.

The system specification is a means of translating the description of needs into a more formal structure and model.

System Design Specification

Nonetheless, every part of the design needs another specification. Specifications can and do exist at various levels as the design is refined and elaborated. Different things must be quantified and at different levels of detail during different phases of the product development. The *System Design Specification* may require that an intersystem communication channel transfer data at the rate of 10,000 bytes per second at a specific bit error rate. The detailed *Hardware* and *Software Specifications* establish the requirements and constraints on their respective components to be able to meet those specifications.

Hardware, Software Specifications

A specification is a precise description of the system that meets stated requirements. Ideally, a specification document should be

- Complete
- Consistent
- Comprehensible
- Traceable to the requirements
- Unambiguous
- Modifiable
- Able to be written

System Specification

The specification should be expressed in as formal a language or notation as possible, yet readable. Ideally, it should also be executable. A *System Specification* should focus precisely on the system itself. It should provide a complete description of its externally visible characteristics, that is, its public interface. External visibility clearly separates those aspects that are functionally visible to the environment in which the system operates from those aspects of the system that reflect its internal structure.

9.9 PARTITIONING AND DECOMPOSING A SYSTEM

System Design Specification

At this point in the design cycle, all of the system requirements have been identified, captured, and formalized into the *System Design Specification*. The next step is to move inside the system and begin the process of specifying and designing the functionality that gives rise to the external behavior.

Throughout all of the previous discussions, modularity and encapsulation have been repeatedly stressed. We will look first at why such an approach is recommended and then at what should be considered as the process of decomposing and ultimately partitioning the system into hardware and software modules proceeds.

9.9.1 Initial Thoughts

So, let's get to the first question, "Why do we do this?" Reuse is one important reason. With each new design, one should always look to the previous project as well as the next one. What can be used from the last project to expedite the development of this one? How can the current design be implemented to support a future feature? Can parts of this design be used in future projects?

Second, many compilers generate object code in segments, one for each module. Such actions may place size restrictions on the individual modules. Poor module builds can significantly affect memory accesses, increase cache misses, promote thrashing, and significantly reduce performance.

Third, often, work assignments are made on a module-by-module basis. Module boundaries should be defined so as to minimize interfaces among different parts of the system. Such a practice simplifies the process of subcontracting some of the work as well. Security issues also play a role when subcontracting is considered. Whether working for a toy company or on a sensitive government project, one needs to consider what information to make available to outside vendors. By properly decomposing a system, the portions that can be outsourced and those for which control over should be retained can be more easily identified.