PART 1 HARDWARE AND SOFTWARE INFRASTRUCTURE

Chapter 1

The Hardware Side – Part 1: An Introduction

THINGS TO LOOK FOR …

- The differences between microprocessors, microcomputers, and microcontrollers.
- The four major functional blocks of a computer and their interconnecting busses.
- How to represent numbers, characters, addresses and instructions in a digital system.
- Different instruction formats and addressing modes.
- Data and control flow through a computer.
- The instruction set architecture level (ISA) model of a computer.
- The computer instruction cycle.
- The register transfer level (RTL) model of a computer.

1.1 INTRODUCTION

Our brief introduction to embedded systems in the Foreword shows that hardware, software, and firmware are essential elements in today's embedded systems. The digital hardware provides the platform from which the three can synergistically perform amazing tasks.

Embedded Systems: A Contemporary Design Tool, Second Edition. James K. Peckol.

Companion website: www.wiley.com/college/peckol

What is to be implemented in hardware and in software or firmware changes with every design – perhaps even within a single design as the requirements and the development evolve. Hardware brings a variety of strengths and weaknesses to the design; software and firmware do the same. As we learn to develop embedded applications, we will learn all their strengths and weaknesses. We will also learn when and how to choose which to use in a design.

In this chapter we will begin with the high-level structure and components: the hardware and computing core of an embedded application. That core is usually manifest as a microprocessor, microcomputer, or microcontroller. At the opposite end of the system hierarchy, we will take our first look at the bits, bytes, and volts as we study how the various and essential kinds of information (numbers, characters, addresses, and instructions) are represented within a digital system. Building on the instructions, we will introduce and study the instruction set architecture (ISA) level and register transfer level (RTL) of the computer. Throughout the remaining chapters of this book, we will develop and study each of these parts of an embedded design in detail. That study will also include the hardware and software interaction, for without both we cannot build any kind of system today.

VLSI ***FPGAs*** ***(C)PLDs, ASICs*** ***MSI*** ***SSI***

In today's high-tech and changing world, we can put together a working hierarchy of hardware components. At the top, we find *VLSI* (Very Large-Scale Integrated) circuits comprising significant pieces of functionality: microprocessors, microcontrollers, *FPGAs* (Field Programmable Gate Arrays), (*C)PLDs* ((Complex) Programmable Logic Devices), and *ASICs* (Application Specific Integrated Circuits). Perhaps we could include memories as well. At the next level down, we find *MSI* (Medium-Scale Integrated) circuits, which bring smaller, yet complete, pieces of functionality. Going down one more step, we have *SSI* (Small-Scale Integrated) circuits. At the very bottom, we have the electrical signals that we use to represent our data and control information and the other signals that come into our system as noise or other unwanted signals. We will develop the hardware side of the design according to that hierarchy.

glue logic

Today, we collect components in the last two categories of integrated circuits (MSI and SSI) into what we call *glue logic*. As we continue to make significant advances in the design and development of more complex digital components, one must wonder about the remaining lifetime of the glue logic components. Tomorrow, the AND and OR gates, as stand-alone entities, may only be available at the Smithsonian or the British or Deutsches museums.

We will start at the core microprocessor level and then look inside the hardware components through a review of the fundamentals of Boolean algebra, finite state machines, as well as arithmetic and logical circuits. In our study of the hardware side, we will study good design practices and some important considerations when developing hardware foundations that are robust, reliable, and maintainable. The complexity of today's systems precludes many of the approaches we used yesterday. Building a breadboard of a design comprising 500 000 gates is neither feasible nor reasonable. At the same time, building a computer model of such a system is entirely practical. Throughout our studies on the hardware side, we will utilize the Verilog modeling language to enable us to test, confirm, and demonstrate the viability of our designs prior to committing to hardware. The language will enable us to work at various levels of detail – at the top or behavioral level, we can confirm high-level functionality, and at the lower level or structural level, we can confirm details of timing, scheduling, and control. Facility at both levels is essential today.

If you already feel comfortable with hardware design and developing hardware systems, take a few minutes to scan through the next several chapters and perhaps review the material on good design practices. If hardware design is new to you, working through the material in this chapter should get you started on the road to digital proficiency. Good luck and have fun.

1.2 THE HARDWARE SIDE – GETTING STARTED

Our study of the hardware side of embedded systems begins with a high-level view of the computing core of the system. We will expand and refine that view to include a detailed discussion of the hardware (and its interaction with the software) both inside and outside of that core. Figure 1.1 illustrates the sequence we will follow.

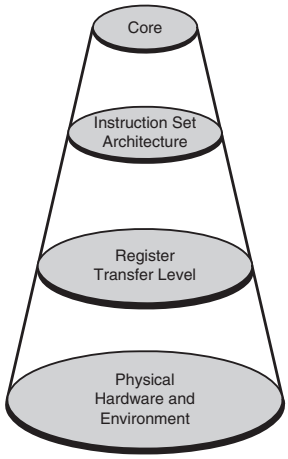

Figure 1.1 Exploring Embedded Systems

The computing core is the central hardware component in any modern embedded application. It often appears as a microprocessor, microcomputer, or microcontroller. Occasionally, it may appear as a custom-designed VLSI circuit or FPGA. It interacts with and utilizes the remaining components of the system to implement the required application. Such actions are under the control of a set of software and firmware instructions. Information and data come into the system from the surrounding environment and from the application. These data are processed according to the software instructions into signals that are sent back out of the system to the application. The software and firmware instructions, as well as signals coming into or going out of the system, are stored in memory.

1.3 THE CORE LEVEL

input, output, memory, datapath, control

At the top, we begin with a model comprising four major functional blocks (*input, output, memory,* and *datapath* and *control*) depicting the embedded hardware core and the high-level signal flow. These are illustrated in Figure 1.2. While there is nothing inherent

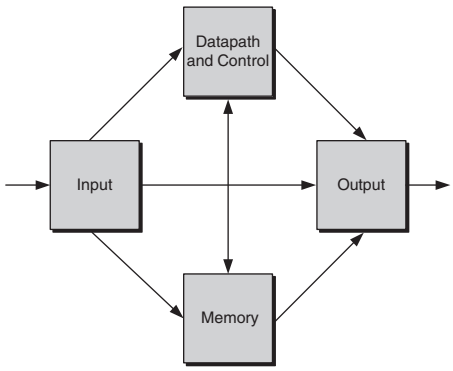

Figure 1.2 Four Major Blocks of an Embedded Hardware Core

in the model that demands a microprocessor, typically, one is used for the computation and control function.

memory
software, firmware

The *memory* block serves to hold collections of program instructions that we call *software* and *firmware*, as well as to provide short-term storage for input data, output data, and intermediate results of computations. Data as well as other kinds of signals come into the system from the external world through the *input* block. Once inside of the system, they may be directed to any number of destinations. The *output* block provides the means to send data or other signals back to the outside world. The *datapath and control* block, more commonly known as the *central processing unit* (CPU), coordinates the activities of the system as well as performing the computations and data manipulation operations necessary to execute the application. In performing its responsibilities, the CPU fetches instructions from memory, interprets them, and then performs the task indicated by the instruction. In doing so, it may retrieve additional data from memory or from the input block. Often, it will also produce information that is sent out of the system.

input
output
datapath and control
CPU
central processing unit

busses

We move signals into, out of, or throughout the system on paths called *busses*. In their most common implementation, busses are simply collections of wires that are carrying related electrical signals from one place to another. We use the term *bus* so that we can speak of such a collection or group as a single entity. Signals flowing on the wires making up the busses are classified into three major categories: *address, data,* and *control*. The data are the key signals that are being moved around; the address signals identify where the data is coming from and where it is going to; and the control signals specify and coordinate how the data is transported.

address, data, control

Think of the arrangement as being similar to your telephone. The number you dial is the *address* of where your conversation will be directed, and the ring is one of the *control* signals indicating to the person you are calling that a call is coming in. Finally, your voice or text message is the *data* that you are moving from your location to the person on the other telephone. As with your telephone, the medium carrying the signal may take many forms: copper wire, fiber-optic cable, or electromagnetic waves.

bits

In the digital world, signals are expressed as collections of binary 0's and 1's; the elements of such collections, the 0's and 1's, are called *bits*. A bit is simply a variable that takes on either of two values. At the hardware level, a bit may be represented by an electrical signal: a binary 0 as 0 V and a binary 1 as 5 V. In an optical communications channel, a bit may also be expressed by the presence or absence of light.

width

The *width* of a bus, that is, the number of signals or bits that it can carry simultaneously, provides an indirect measure of how quickly information can be moved. Transferring 64 bits of data on a bus that is 32 bits wide requires two transfers to move the data. In contrast, a bus that is only 8 bits wide will require eight transfers. Figure 1.3 illustrates moving such

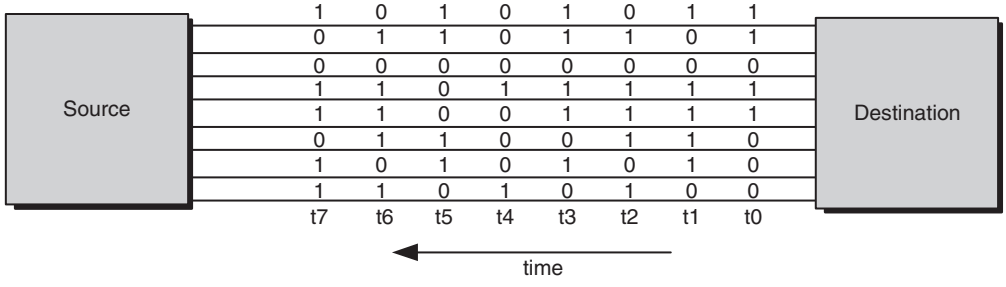

Figure 1.3 Data Movement Over an 8-Bit Bus

a set of data over an 8-bit bus from a source module to a destination module. In the model, each transfer requires one time unit; the process begins at time t0 and completes at time t7. Time increases to the left.

The following C code fragment might produce such a pattern,

```
for (i = 0; i < 8; i++)
{
    printf("%i", a[i]);
}
```

The source of the transfer is the array of eight bit values; the destination is perhaps a display. In Figure 1.4, we refine the high-level functional diagram to illustrate a typical bus configuration comprising the address, data, and control lines.

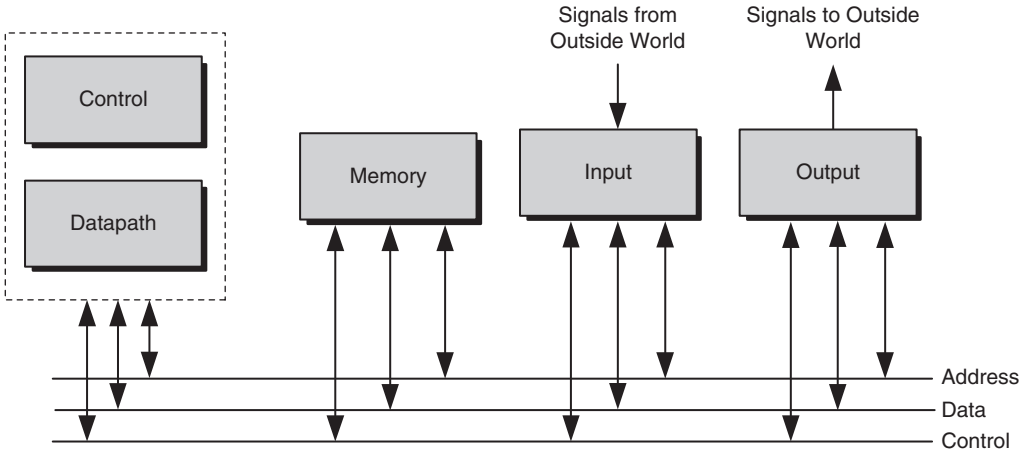

Figure 1.4 A Typical Bus Structure Comprising Address, Data, and Control Signals

bus width, address bus data bus, control bus

None of the busses is required to have the same number of lines. To avoid cluttering a drawing by including all of the signals or conducting paths that make up a bus, we will often label the bus width using the annotation */bus width*, as illustrated in Figure 1.5. In this example, the *address bus* is shown to have 18 signals, the *data bus* 16, and the *control bus* 7.

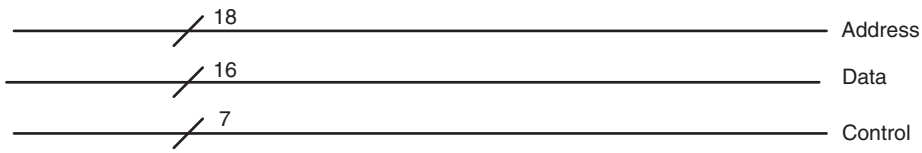

Figure 1.5 Identifying the Number of Signals in a Bus

In practice, such a block diagram will be implemented using a microprocessor, microcomputer, or microcontroller. Let's look at each of these, beginning with the microprocessor, with the goal of understanding the high-level structure of each and the differences among them.

1.3.1 The Microprocessor

A *microprocessor* is an integrated implementation of the CPU portion (control and arithmetic and logical unit) of the machine; it is often simply referred to as a CPU or datapath. Microprocessors differ in complexity, power consumption, and cost. Today, microprocessors range from devices with only a few thousand transistors at a cost of a dollar or less to units with 5–10 million transistors and a cost of several thousand dollars.

internal registers
registers

One may also find differences in the internal architecture of the machine, including the number of *internal registers*, the overall control structure of the machine, and the internal bus structure. *Registers* are small amounts of high-speed memory that are used to temporarily store frequently used values, such as a loop index or the index into a buffer. Increasingly, the internal single-memory scheme that characterizes the von Neumann machine is giving way to the Harvard architecture and the benefits of simultaneous instruction and data access.

To implement a complete (embedded) computer system, we must still include the input/output subsystems and the external (to the microprocessor) memory system. We also include a clock or timing reference as the basis for timing, scheduling, or measuring elapsed time. All such components are connected via a system bus or busses. Figure 1.6 depicts a high-level block diagram of a microprocessor-based system.

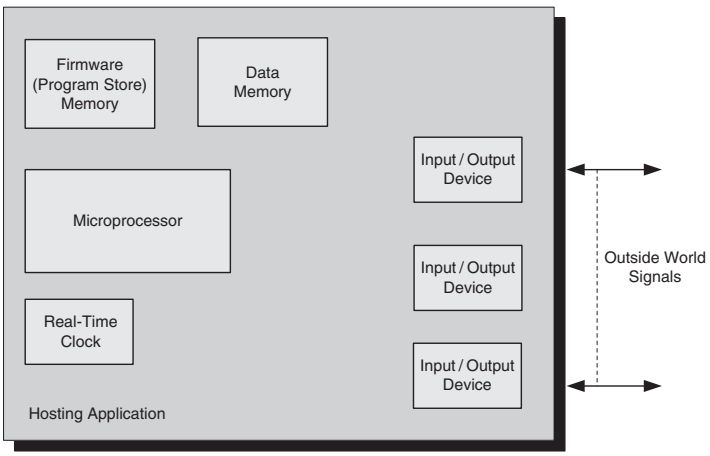

Figure 1.6 A Block Diagram for a Microprocessor-Based System

firmware
data store

Read Only Memory
ROM
Random Access Memory
(RAM)

External to the microprocessor, we see two different memory blocks. The *firmware*, or program store, contains the application code, and the *data store* contains data that are being manipulated, sent to, or brought in from the external world. In the embedded world, we refer to the application code as firmware because it is generally stored in a *Read Only Memory* (ROM), rather than on a hard drive as one might do for a desktop application. The data memory is usually made up of *Random Access Memory* (RAM).

Caution: The two separate pieces of memory do not change the architecture from von Neumann to Harvard unless two separate busses are connecting them to the processor.

1.3.2 The Microcomputer

microcomputer

The *microcomputer* is a complete computer system that uses a microprocessor as its computational core. Typically, a microcomputer will also utilize numerous other large-scale integrated (LSI) circuits to provide necessary peripheral functionality. As with the microprocessor, the complexity of microcomputers varies from simple units that are implemented on a single chip along with a small amount of on-chip memory and elementary I/O system to the complex that will augment the microprocessor with a wide array of powerful peripheral support circuitry. Costs, of course, are commensurate with capability.

1.3.3 The Microcontroller

microcontroller

The *microcontroller*, as illustrated in Figure 1.7, brings together the microprocessor core and a rich collection of peripherals and I/O capability into a single integrated circuit. Such additions typically include timers, analog-to-digital converters, digital-to-analog converters, digital I/O, serial or parallel communications channels, and direct memory access (DMA). A memory subsystem may or may not be included. If the memory is not included, the designer must add such capability outside of the microcontroller. Microcontrollers find great utility in basic embedded applications where low cost is a significant constraint.

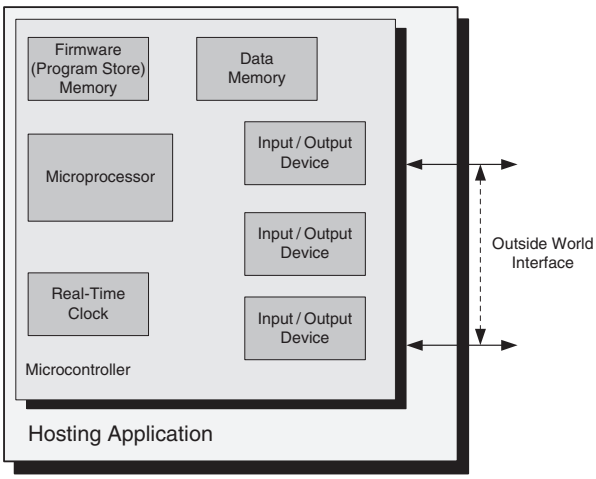

Figure 1.7 A Block Diagram for a Microcontroller-Based System

We see that we have the same components as we found in the microprocessor-based system. However, now they are integrated into a single unit.

1.3.4 The Digital Signal Processor

digital signal processor (DSP)

In addition to the three different types of general purpose computing engines that we have discussed, a special purpose microprocessor called a *digital signal processor* (DSP) is becoming increasingly common in embedded applications. The DSP is typically used in

conjunction with a general purpose processor to perform specialized tasks such as image, speech, audio, or video processing. A representative block diagram for a DSP is given in Figure 1.8.

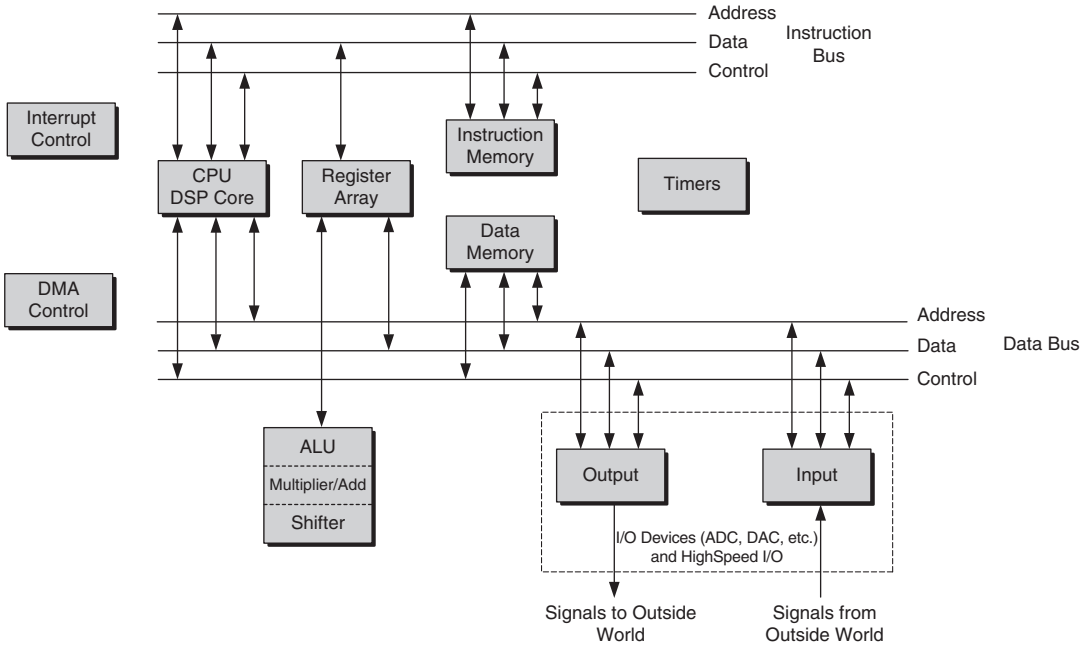

Figure 1.8 A Block Diagram for a Digital Signal Processor

The tasks performed by the DSP often require it to interface with the analog world. Real-world analog signals are captured through an analog-to-digital converter, processed, and returned through a digital-to-analog converter. One of the major strengths of the DSP is its ability to perform basic arithmetic computations, such as multiply, add, and shift, at the rate of millions of operations per second. To support high-speed arithmetic, the device will often implement a multiply-accumulate (MAC) primitive in which a multiply and add to the accumulator is performed in a single operation, which is useful in matrix operations. Its arithmetic operations often utilize saturation arithmetic in which overflows (underflows) remain at the maximum (minimum) value rather than wrapping around. In further support of high-speed signal processing, the DSP device is architected as a Harvard rather than the von Neumann machine and incorporates multiple computational units, a large number of registers, and wide high-bandwidth data busses.

Before proceeding to the next level of detail in our study of the hardware, let's move to the opposite end of the hierarchy and examine some of the signals that we are moving among the various components as well as into and out of the system.

1.4 REPRESENTING INFORMATION

In any embedded application, in addition to the expected numbers and symbols or characters, we must be able to represent both firmware instructions and the data that such

instructions may be operating on. Because these instructions are stored in memory, we also need to be able to represent addresses in memory where the data and instructions are stored. In the next several sections, we will briefly examine how we can represent these different kinds of information and, with such representations, what limitations we will encounter.

1.4.1 Word Size

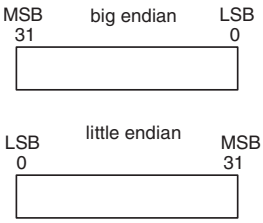

Figure 1.9 Big Endian vs. Little Endian Notation

One of the terms that we use to characterize a microprocessor is its word size. Generally, the word size in a microprocessor refers to the size of an integer. We will assume for the remainder of this study that we are working with a microprocessor that is using a word size of 32 bits. Such a processor is called a 32-bit machine. For such a system, we can interpret an unsigned integer, for example, according to either of the representations that we see in Figure 1.9.

We know that the data bus will be 32 bits wide and that each transfer on the bus will take one unit of time. If we examine the format of the word, several things are evident. The word consists of four bytes. The bits can be interpreted so as to place the most significant byte on the left and the least significant on the right, or vice versa.

> ***Important Point***
>
> The interpretation of the order of the bits is just that, an interpretation. There is nothing inherent in the construction of a number that dictates which end has the most or least significant bits.

big endian
little endian

A word interpreted as in the top part of Figure 1.9 is said to be written in *big endian* format, and one written as in the lower figure is said to be written in *little endian* format. Different microprocessors, operating systems, and networks interpret such words in different ways. When executing a design, it is absolutely essential to determine which format each of these components in the system uses. In this text, we will assume a big endian interpretation unless specified otherwise.

1.5 UNDERSTANDING NUMBERS

We have seen that within a microprocessor, we do not have an unbounded number of bits with which to express the various kinds of numeric information that we will be working with in an embedded application. The limitations of finite word size can have unintended consequences for the results of any mathematical operations that we might need to perform. Let's examine the effects of finite word size on resolution, accuracy, errors, and the propagation of errors in these operations. In an embedded system, the integers and floating point numbers are normally represented as binary values and are stored either in memory or in registers – small pieces of memory. The expressive power of any number is dependent on the number of bits in the number. Although it is certainly true that, in the extreme, a number can be stored and transferred as a collection of words of size one, such an implementation is neither practical nor efficient. At the end of the day, we have memory limitations as well.

1.5.1 Resolution

To begin to understand the problem, let's consider a 4-bit word. If the word is used to hold a number, the bits comprising the word can be interpreted in several ways as presented in Table 1.1.

Table 1.1 Interpreting a Four-Bit Number

Interpretation	Expressive power
Integer	0–15
Real	
xxx.x	0–7.5
xx.xx	0–2.75
x.xxx	0–1.6875

If the bits are interpreted as expressing an unsigned integer, that integer may range from 0 to 15; the resolution is 2^0. Interpreting the number as a real with 2 bits devoted to the fractional component provides *two digits of resolution*. That is, we can express and resolve a binary number to 2^{-2}. The meaning of such a limitation is seen in the following example.

two digits of resolution

EXAMPLE 1.1

To represent the number 2.3 using a 4-bit binary number with 2 bits of resolution, the best that one can write is either (10.10) 2.5 or (10.01) 2.25. The error in the expression will be either +0.2 or −0.05. All that can be resolved is ±0.25.

Because word size limits one's ability to express numbers, eventually, we are going to have to either round or truncate a number in order to be able to store it in internal memory. Thus, faced with truncation or rounding, one can ask which provides the greater accuracy, and which will give the best representation of a measured value? Which alternative is more or less accurate?

Let's consider a real number, N. Following either truncation or rounding of the original number to fit the microprocessor's word size, the number will have a fractional component of n bits. The value of the least significant bit is 2^{-n}. Whether we round or truncate, the resulting number will have an error. The graphs in Figure 1.10 plot the original number versus the truncated or rounded number.

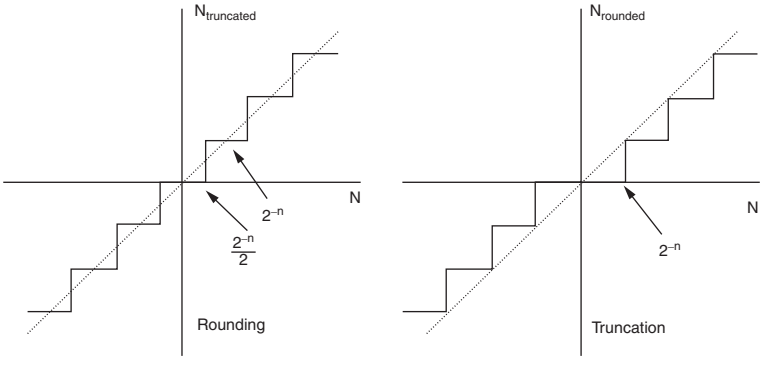

Figure 1.10 Truncation vs. Rounding

The error following the operation is computed as

$$E_R = N_{rounded} - N \tag{1.1}$$

$$E_T = N_{truncate} - N \tag{1.2}$$

and given in Table 1.2.

Table 1.2 Truncation vs. Rounding Error

	N	$N_{rounded}$	$N_{truncated}$	Error
Truncation	0		0	0
	2^{-n}		0	-2^{-n}
Rounding	0	0		0
	$½\ 2^{-n}-$	0		$½\ 2^{-n}-$
	$½\ 2^{-n}+$	2^{-n}		$½\ 2^{-n}$

As the graph and table illustrate, the operations produce the following ranges of errors

Truncation

$$-2^{-n} < E_T \leq 0$$

Rounding

$$-\frac{1}{2}2^{-n} < E_R \leq \frac{1}{2}2^{-n}$$

Observe that the full range of the error is the same; however, for the case of rounding, the error is more evenly distributed and the maximum error is less.

1.5.2 Propagation of Error

Next, we analyze how the errors propagate under processing. We begin with two perfect numbers, N_1 and N_2. Under truncation, the error is less than 1 least significant bit.

1.5.2.1 Addition

We can express the numbers with an error as

$$N_{1E} = N_1 + E_1 \tag{1.3}$$

$$N_{2E} = N_2 + E_2 \tag{1.4}$$

$$\begin{aligned} N_{1E} + N_{2E} &= (N_1 + E_1) + (N_2 + E_2) \\ &= N_1 + N_2 + E_1 + E_2 \end{aligned} \tag{1.5}$$

The error in the resulting sum is in the range

$$2 \bullet 2^{-n} < E_T \leq 0 \Rightarrow 2^{1-n} < E_T \leq 0$$

Observe that the resulting error is the (algebraic) sum of the original errors.

1.5.2.2 Multiplication

We can express the numbers with an error as

$$N_{1E} = N_1 + E_1 \tag{1.6}$$

$$N_{2E} = N_2 + E_2 \tag{1.7}$$

$$\begin{aligned} N_{1E} \bullet N_{2E} &= (N_1 + E_1) \bullet (N_2 + E_2) \\ &= (N_1 \bullet N_2) + (N_2 \bullet E_1 + N_1 \bullet E_2) + (E_1 \bullet E_2) \end{aligned} \tag{1.8}$$

If we neglect $E_1 \bullet E_2$, the resulting error is

$$(N_2 \bullet E_1 + N_1 \bullet E_2) < E_T \leq 0$$

Observe that the magnitude of the error now depends on the size of the numbers.

To further illustrate the propagation of error in basic calculations, consider the measurement system in Figure 1.11 that is designed to determine power in a resistive load. The power in the resistor is computed from measurements of the voltage drop across the resistor and the current flow through the part. Those measurements are given as

$$E = 100\,\text{VDC} \pm 1\%$$
$$I = 10\,\text{A} \pm 1\%$$
$$R = 10\,\Omega \pm 1\%$$

The power dissipated in the resistor, R, can be calculated in three ways. In theory, they should produce identical results.

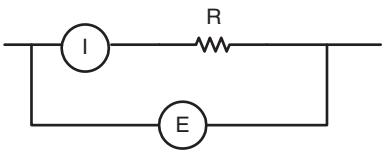

Figure 1.11 A Simple Measurement System

In each of the three computations, lower order terms are neglected; however, doing so will have minimal effect on the final results.

$$\begin{aligned} EI &= (100V \pm 1\%) \cdot (10A \pm 1\%) \\ &= ((1000 \pm 10 \cdot 1\%)) \pm ((100 \cdot 1\%) \pm (1\% \cdot 1\%)) \\ &= (1000 \pm 1.1) \\ EI &\Rightarrow 998.9 \rightarrow 1001.1 \end{aligned} \tag{1.9}$$

$$\begin{aligned} I^2R &= (10A \pm 1\%) \cdot (10A \pm 1\%) \cdot (10\Omega \pm 1\%) \\ &= (100 \pm (20 \cdot 1\%) \pm (1\% \cdot 1\%)) \cdot (10 \pm 1\%) \\ &= (100 \pm 0.2) \cdot (10 \pm 1\%) \\ &= ((1000 \pm 2) \pm ((100 \cdot 1\%) \pm (0.2 \cdot 1\%))) \\ &= (1000 \pm 3) \\ I^2R &\Rightarrow 997 \rightarrow 1003 \end{aligned} \tag{1.10}$$

$$\begin{aligned} \frac{E^2}{R} &= \frac{(100V \pm 1\%) \cdot (100V \pm 1\%)}{(10\Omega \pm 1\%)} \\ &= \frac{(10000 \pm 2) \cdot (1\% \pm 1\%)}{(10 \pm 1\%)} \\ \frac{E^2}{R} &\Rightarrow 908.9 \rightarrow 1111.3 \end{aligned} \tag{1.11}$$

The results of the three calculations not only yield three different answers, but, depending on which formula is used, have substantially differing error magnitudes as well.

These simple examples illustrate rather graphically that when one is performing mathematical computations, it is important to understand where errors can arise, how they can propagate under mathematical operations, and how such phenomena can affect any calculations. Such errors can have serious consequences for the safety of any applications if we are not careful.

1.6 ADDRESSES

In the earlier functional diagram as well as in the block diagram for a microprocessor, we learned that information is stored in memory. Each location in memory has an associated address much like an index in an array. If an array has 16 locations to hold information, it will have 16 indices. If a memory has 16 locations to store information, it will have 16 addresses. Information is accessed in memory by giving its address. As we found with encoded characters, each address has a unique binary pattern. Addresses begin at binary 0 and range to the maximum value the word size will permit.

For a word size of 32 bits, the addresses will range (in hex) from 00000000 to FFFFFFFF. Thus, with 32 bits, we have up to 4 294 967 296 unique combinations and, therefore, that same number of possible addresses. Of course, we may not use them all, but they are there. Figure 1.12 illustrates how a word might look if the bits are interpreted as expressing an address. In fact, an address does not look any different from an unsigned integer, which, in reality, it is. It is important that it is not a signed integer. The microprocessor does not support negative addresses.

The following C or C++ declarations place the integer value 10 in binary into some location in memory. Let's say at address 3000.

```
int myVar = 10;
int* myVarPtr = &myVar; // take the address of myVar
                        // assign it to the pointer variable myVarPtr
```

myVar ***value***

When interpreted by the system, the code fragment directs the system to set aside another memory word to hold the address of the signed integer, *myVar* (let's say at address 5000) and puts 3000 into that address. The *value* at address 5000 points to address 3000. The accompanying diagram in Figure 1.13 illustrates such an arrangement.

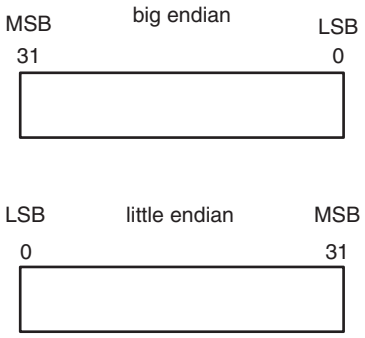

Figure 1.12 Expressing Addresses

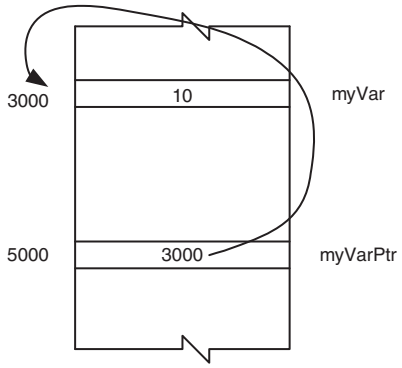

Figure 1.13 Using the Value of One Variable to Hold the Address of Another Variable

1.7 INSTRUCTIONS

The last view that we want to take on a word is as an instruction. Our examination at this stage of our study will be at a very high level. As we proceed with our studies, we will examine instructions in greater detail.

actions

To start, we stipulate that the purpose of an instruction is to direct the hardware of the microprocessor to perform a series of *actions*. Such actions can include the following: perform an arithmetic or logical calculation, assign or read the value of a variable, or move data from one place to another such as from input to memory or from memory to output.

operations

In the parlance of instructions, such actions are called *operations*.

operands
arity

The entities that instructions operate on are denoted *operands*. The number of operands that an instruction operates on at any time is called the *arity* of the operation.

Let's look at the following code fragment:

```
x = y + z;
```

binary

Here we have two operations, the addition operation and the assignment operation. Firstly, the addition operation: that operation is performed on two operands, *y* and *z*. The addition operator is said to be a *binary* operator – its arity is two. Now, the assignment operator: the operation is performed by giving *x* the value returned by the addition operation. It also is performed on two operands, the result of the addition and the operand *x* – its arity is two as well.

binary operators
unary operators

In C and C++, both operators are referred to as *binary operators*. Operators taking only a single operand have an arity of one and are referred to as *unary operators*. With one exception, all operators in C and C++ are either unary or binary.

operation
operands

We see, then, that an instruction must contain at least two components, the *operation*, and the *operands*. Depending on the arity of the operation, the instruction may have one or two operands. Let's look at several common C or C++ instructions.

1. `x = y;`

two operands

The instruction expresses the basic C/C++ assignment *operation* in which the value of the *operand y* (the source operand) is assigned to the *operand x* (the destination operand).

binary operator
two-operand instruction,
two-address instruction

Analyzing the format of the instruction, we see that we have two operands, *x* and *y*, thus making the operator a binary operator. Such an instruction is thus referred to as a two-operand or two-address instruction.

2. `z = x + y;`

The code fragment is adding the two operands, *x* and *y*; the result is assigned to the operand *z*. Analyzing, we see that we have two operations: an addition operation and an assignment operation. Both are binary.

For the addition operation, the operands *x* and *y* are the sources, and the (temporary) result is the destination. Moving to the assignment operation, the result (destination) of the addition is the source of the assignment operation. The operand *z* is the destination of the assignment.

three operands
three-operand instruction,
three-address instruction

If we ignore the transient intermediate result, we see that for the code fragment, we have three operands, *x* and *y* are sources and *z* is a destination. Such an instruction is designated a three-operand or three-address instruction.

3. `x = x + y;`

The code fragment is adding the two operands, *x* and *y*. The result is assigned to the operand *x*. Analyzing, once again, we see that we have two operations: an addition operation and an assignment operation. Both are binary.

As before, for the addition operation, the operands *x* and *y* are the sources, and the (temporary) result is the destination. Moving to the assignment operation, the destination of the temporary result from the addition is the source of the assignment operation. The operand *x*, in addition to being one of the source operands, is also the destination of the assignment.

We see that one operand serves the dual role of source and destination. In that role, the value that it held as the source is lost as a result of the assignment.

operation
one-operand
one-operand instruction
one-address instruction

If we ignore the transient intermediate result as before, we see that for the code fragment, we have two operands, *x* and *y* are sources and *x* is also a destination. Such an instruction is designated a two-operand or two-address instruction.

4. `++x or x++;`

In the code fragment, the requested operation is to increment the value of the variable. In this case, the variable *x* is both the source and the destination of the operation; we have only one operand. Such an instruction is designated as a one-operand or one-address instruction.

one-, two-, three-operand instruction

The previous code fragments have illustrated three classes of instructions we might find in the system software or firmware. These classes are the *one-, two-,* or *three-operand instruction*. Let's now see how we can interpret the bits in a 32-bit word to reflect such instructions. Any such interpretation will have to support the ability to express both operands and operations as seen in Figure 1.14.

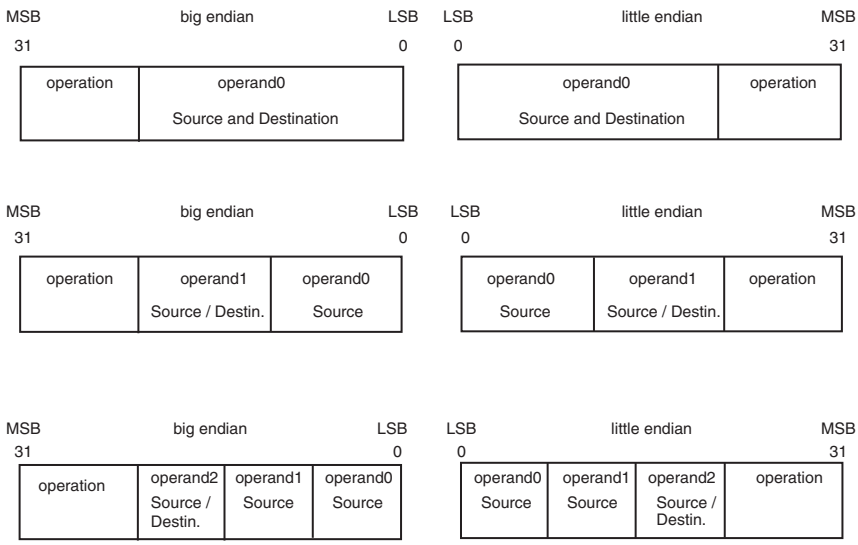

Figure 1.14 Expressing Instructions

groups
fields, operation
operands

In Figure 1.14 we see that within the 32-bit word the bits are aggregated into *groups* or *fields*. Some of the fields are interpreted as the *operation* to be performed, and others are seen as the *operands* involved in the operation. At this juncture, we have not specified how many bits comprise each field. Such a determination is made during the early stages of the development of the architecture and design of the microprocessor.

The field used to identify the operation is the easiest one to specify the size of. The first step in specifying the size is to decide how many different instructions the microprocessor is to support. Such instructions can include arithmetic operations (e.g. add, subtract, multiply, or divide), logical operations (e.g. and, or, shift, or invert), data movement, or flow of control (e.g. jump, function call, or branch). Once the number of instructions is determined, then each is assigned a unique code, exactly as was done earlier when encoding characters.

operation code,
op-code

Such a code is called the *operation code* or *op-code*. If the microprocessor is to support 128 instructions, a minimum of 7 bits will be necessary. The designer may elect to allocate 8 bits to permit room to incorporate additional instructions at a later time or in support of an enhanced version of the microprocessor.

1.8 REGISTERS – A FIRST LOOK

The size of the fields allocated to the operands is not much more complex. Before answering the question, however, we must make a slight digression. For those readers who have begun to anticipate a small problem, you are correct. We have been discussing how we can interpret a 32-bit word as various types of data (operands) that can be operated on by user-selected operations. If an instruction, containing even a single operand, in addition to the op-code, is 32 bits, a 32-bit piece of data will not fit into any of the field(s) allocated to hold the operand.

register

To solve this seemingly intractable problem, we utilize a hardware component that is called a *register*. A register is a special kind of memory that is large enough to hold a single data word. A register is a piece of short-term memory that temporarily holds the operands during the execution of an instruction.

Prior to executing the instruction, the operand(s) are moved from memory into registers and then back to memory if the data value is not going to be needed in the immediate future. While such continual movement of data words into and out of memory and into and out of registers seems to involve a lot of extra work, the higher speed of registers compared with the memory we have discussed so far can significantly improve system performance.

Depending on the architecture of the microprocessor, it may have a few registers – 16 to 256 or so – or it may have over 1000. Those microprocessors in the former category are referred to as *Complex Instruction Set Computers* (CISCs), and those in the latter are called *Reduced Instruction Set Computers* (RISCs). While the number of registers is not the only (or most significant) difference between the two architectures, their effect on system performance can be significant.

Complex Instruction
Set Computers (CISC)
Reduced Instruction
Set Computers (RISC)

We can now examine the role that registers play in the format of an instruction. The contents of the operand field within an instruction is not the operand; rather, it is a binary number indicating which of the microprocessor's registers contains the operand.

Let's assume a hypothetical microprocessor with 144 instructions. To permit each instruction to be uniquely identified, we will have to specify that the op-code contains 8 bits since $2^7 < 144 < 2^8$. Let's further assume that the microprocessor is designed to include 256 registers. To permit each register to be uniquely identified will also require 8 bits.

Our earlier diagram for the various instruction formats can now be modified to reflect the new interpretation of the operand fields, as illustrated in Figure 1.15.

The operand fields in the two-and one-operand instructions are large enough to provide more than 256 combinations as register designators; most of the combinations will remain unused.

Figure 1.16 summarizes the big endian interpretations of a word in a microprocessor system. The little endian interpretations follow naturally.

> ***Note***
>
> It is important to understand that an aggregate of bits has no inherent meaning.
>
> Meaning comes from our interpretation of those bits, and this is what is defined as type information.

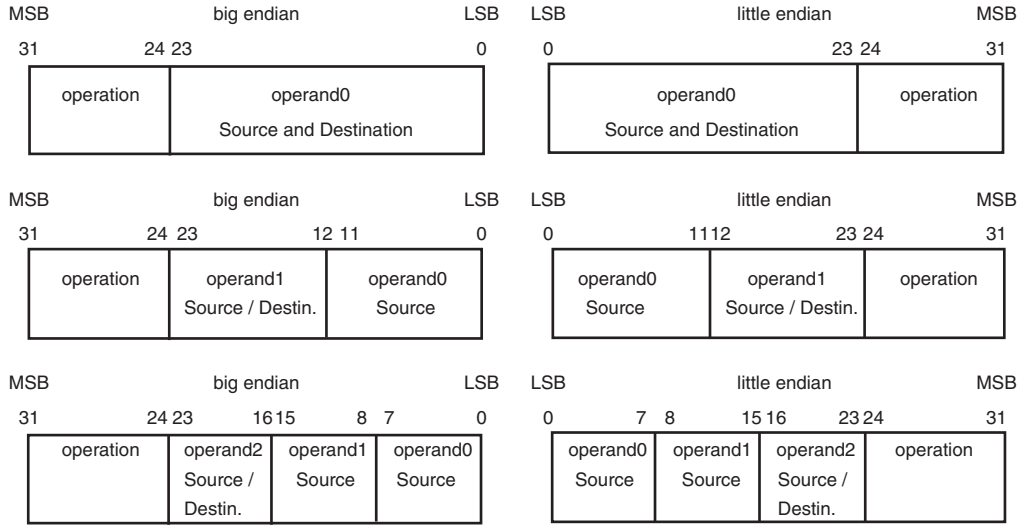

Figure 1.15 Expressing Instructions

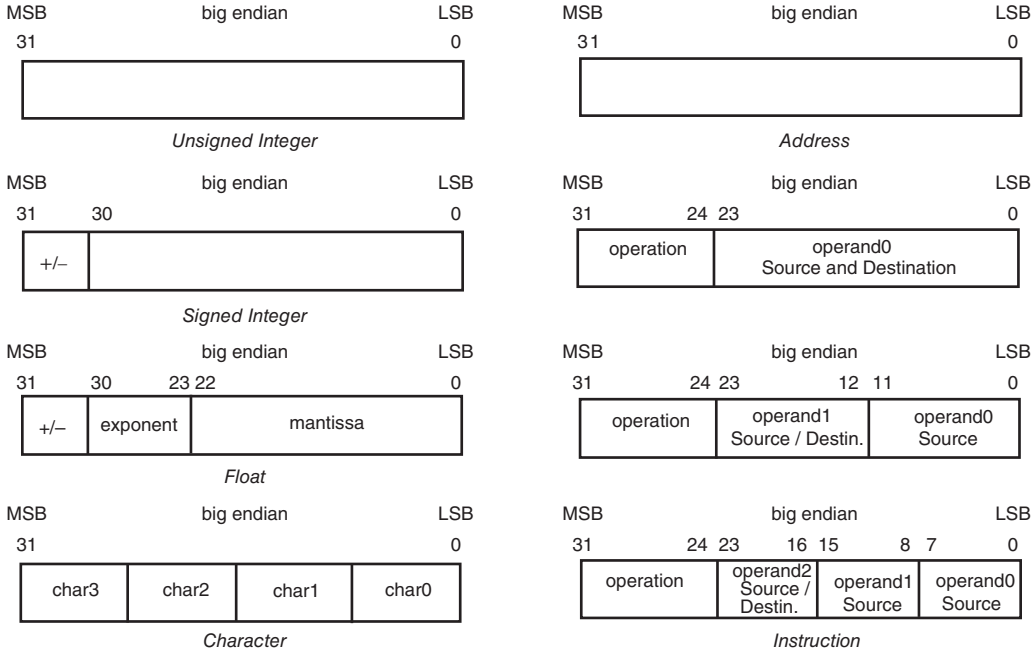

Figure 1.16 Possible Interpretations of a Set of Bits as Big Endian Representation. Little Endian Follows Similarly

1.9 EMBEDDED SYSTEMS – AN INSTRUCTION SET VIEW

assembly language

The software (firmware) in an embedded system is generally written in a high-level language such as C or C++. In other cases, it may be written in what is called the *assembly language* for the machine on which the application is to run. Sometimes combinations of the two are used, as is the case when portions of programs must be optimized for speed or size.

machine language ***instruction set*** ***instruction set architecture***

When working with assembly language, we are working one step removed from the microprocessor's *machine language* – the collection of 0's and 1's that control the hardware components in the execution of an instruction. At the assembly language level, we are working with the set of instructions that the machine supports – its *instruction set*. Such a set drives the architecture, the design of the underlying hardware of the processor. That architecture, the *ISA*, thus provides to the programmer the public interface for the underlying hardware.

machine code ***assembler*** ***IEEE Standard for Microprocessor Assembly Language***

At the assembly language level, mnemonic names are given to binary patterns expressed by the op-codes to make them easier to work with. A program written in the machine's assembly language is translated into *machine code* by a software tool called the *assembler*. Thus, the machine code reflects binary encoding of the machine's instructions or op-codes. Such a set of op-codes for an ISA can be viewed as the machine language for that particular architecture. For the discussion that follows, the assembly language instructions are taken from the *IEEE Standard for Microprocessor Assembly Language* – IEEE Std. 694–1985.

The complete list of assembly language instructions and how to work with them is given in the support manuals for the specific processor. These will be provided by the developer of each processor.

1.9.1 Instruction Set – Instruction Types

transfer ***store, operate, make decisions***

A microprocessor's instruction set specifies the basic operations supported by the machine. From the earlier functional model, we see that the objectives of such operations are to *transfer* or *store* data, to *operate* on data, and to *make decisions* based on the data values or outcome of the operations. Corresponding to such operations, we can classify instructions into the following groups:

- Data Transfer
- Flow of Control
- Arithmetic and Logic

Data transfer instructions provide the means and mechanism for moving data within the system and for executing exchanges with external devices, including memory. The flow of control instructions determine the order in which instructions comprising the embedded application are executed. Such instructions often respond to the side effects resulting from the arithmetic or logical operations. The arithmetic and logical instructions provide the majority of the computational capabilities and useful functionality of a microprocessor.

1.9.2 Data Transfer Instructions

Data transfer instructions are responsible for moving data around inside the processor as well as for bringing data in from the outside world or sending data out. Each such instruction

data, location of the data, source, destination

must have four pieces of information: the *data*, the *location of the data* – the *source* of the transfer, and where the data is to be sent – the *destination* of the transfer.

The source and destination can be any of the following:

- A Register
- Memory
- An Input or Output Port

as is illustrated in Figure 1.17.

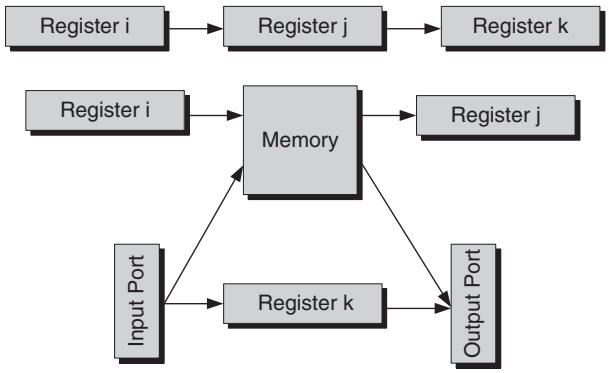

Figure 1.17 Transferring Data

Some of the more common instructions used in support of such data transfers include those presented in Figure 1.18. A processor design will not implement both the *MOVE* and the *LD/ST* pair. For completeness, both are illustrated in the figure.

LD destination, source	Load—source operand transferred to destination operand can be either register or memory location.
ST source, destination	Store—source operand transferred to destination operand source must be a register and the destination must be memory.
MOVE destination, source	Transfer from register to register or memory to memory.
XCH destination, source	Interchange the source and destination operands.
PUSH/POP	Oper and pushed onto or popped off of the stack.
IN/OUT destination, source	Transfer data from or to an input/output port.

Figure 1.18 Data Transfer Instructions

Data transfer is supported by instructions using any of the three different address formats we discussed earlier. The op-code portion of each instruction identifies the operation to be performed on the operands. The path to the actual selection of the operands is controlled by the *addressing mode* specified for the operand.

addressing mode

1.9.2.1 Addressing Modes

Typically, a microprocessor design will implement 4–8 different addressing modes. A portion of each operand field is designated as a specification to the hardware as to how to interpret or use the information in the remaining bits of the associated address field. That specification is called the *address mode* for the operand. The address that is ultimately used to select the operand is called the *effective address*.

addressing mode
effective address

Addressing modes are included in an instruction in order to offer the designer greater flexibility in accessing data and controlling the flow of the program as it executes. However, some of the address variations can impact flow through a program as well as the execution time for individual instructions. We will discuss this issue in greater detail when we examine methods for optimizing the performance of an application. Each is identified by a unique binary bit pattern that is interpreted. The drawings in Figure 1.19 refine our earlier expression of each instruction format to reflect the inclusion of the address mode information.

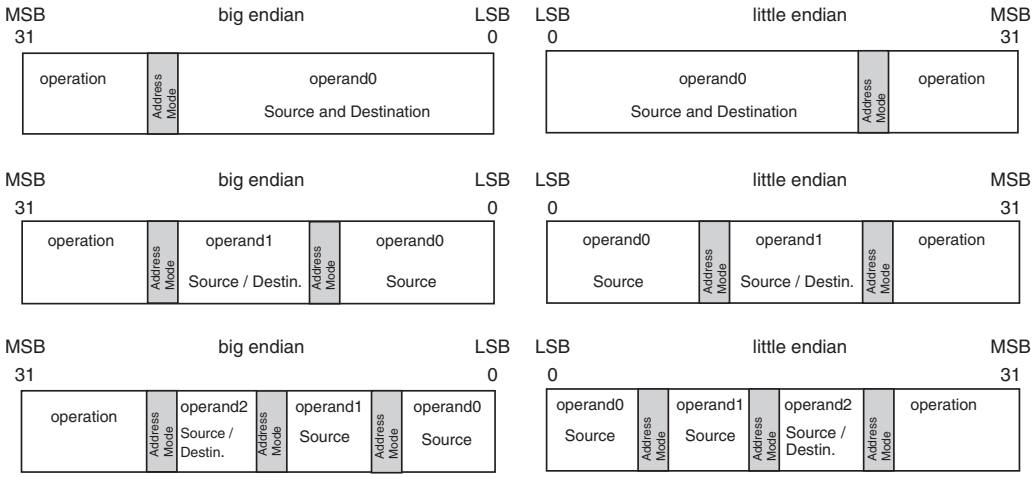

Figure 1.19 Instruction Types Enhanced to Include Address Mode Information

Some of the more commonly used addressing modes include

- Immediate
- Direct and Indirect
- Register Direct and Register Indirect
- Indexed
- Program Counter Relative

address mode field,
operand address

We will examine each of these modes in the upcoming paragraphs. To support the five modes plus the direct/indirect selection, the *address mode field* associated with each *operand address* will have to be 4 bits wide. Let's now examine each of these addressing modes and identify their strengths and weaknesses.

1.9.2.1.1 IMMEDIATE MODE

immediate

An *immediate* mode instruction uses one of the operand fields to hold the value of the operand rather than a reference to it, as shown in Figure 1.20. The major advantage of such an instruction is that the number of memory accesses is reduced. Fetching the instruction retrieves the operand at the same time; there is no need for an additional access. Such a scheme works well if the value of the immediate operand is small, as might be found for loop indices or initializing values.

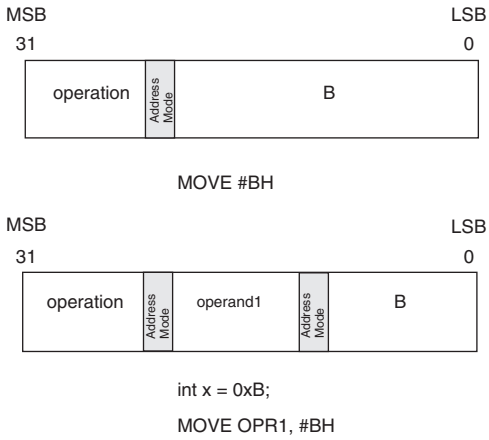

Figure 1.20 Immediate Mode Instruction Formats

arithmetic and logic unit (ALU)

The immediate instruction might appear as a one- or two-operand instruction, as illustrated in Figure 1.20. The one-operand version contains only the immediate value. Without an explicit destination, the target must be implied. Typically, that is the accumulator in the *arithmetic and logic unit (ALU)*. The two-operand version illustrates the operation at both the C or C++ level and the assembly language level. In the former case, the variable y is declared and initialized to the hex value 0×B. Such an expression is compiled into an assembly language statement of the kind shown.

The immediate value in the assembly language expression is intended to be a hex number and is so designated by the H suffix on the number. The first form of the instruction has the accumulator in the *ALU* as an implied destination; there is no C or C++ level equivalent. After all, the developer is not supposed to be aware of the accumulator at such a level. The second form sets the value of operand1 to the hex value B.

On some processors, the instruction mnemonic designates that the operation is to use an immediate operand. In such cases, the instruction may be written as illustrated in Figure 1.21.

```
STI          - Store immediate
LDI / LOADI  - Load Immediate
MOVI         - Move Immediate
```

Figure 1.21 Variations on the Immediate Mode Instruction

1.9.2.1.2 DIRECT AND INDIRECT MODES

direct, indirect

When using the *direct* and *indirect* addressing modes, we are working with operand addresses rather than operand values. In both cases, the first level of address information

is contained in the instruction. The difference between the two modes is that, in the direct mode, the contents of the operand field in the instruction are the address of the desired operand, whereas in the indirect case the field contains the address of the address of the operand. In the latter case, the operand is fetched through one level of indirection.

Because of the limited size of an operand field, the range of memory locations that can be addressed is less than what is possible with the register direct and indirect modes discussed next. With either mode, the major disadvantage is the additional memory accesses necessary to retrieve an operand.

In Figure 1.22, two different data transfer operations are shown. For the *direct* operation, at the C/C++ level, the value pointed to by one variable, *yPtr,* is assigned to a second variable pointed to by *xPtr*. At the assembly language level, the *MOVE* instruction directs that the contents referenced by operand1 be copied to the location referenced by operand0.

direct, yPtr
xPtr

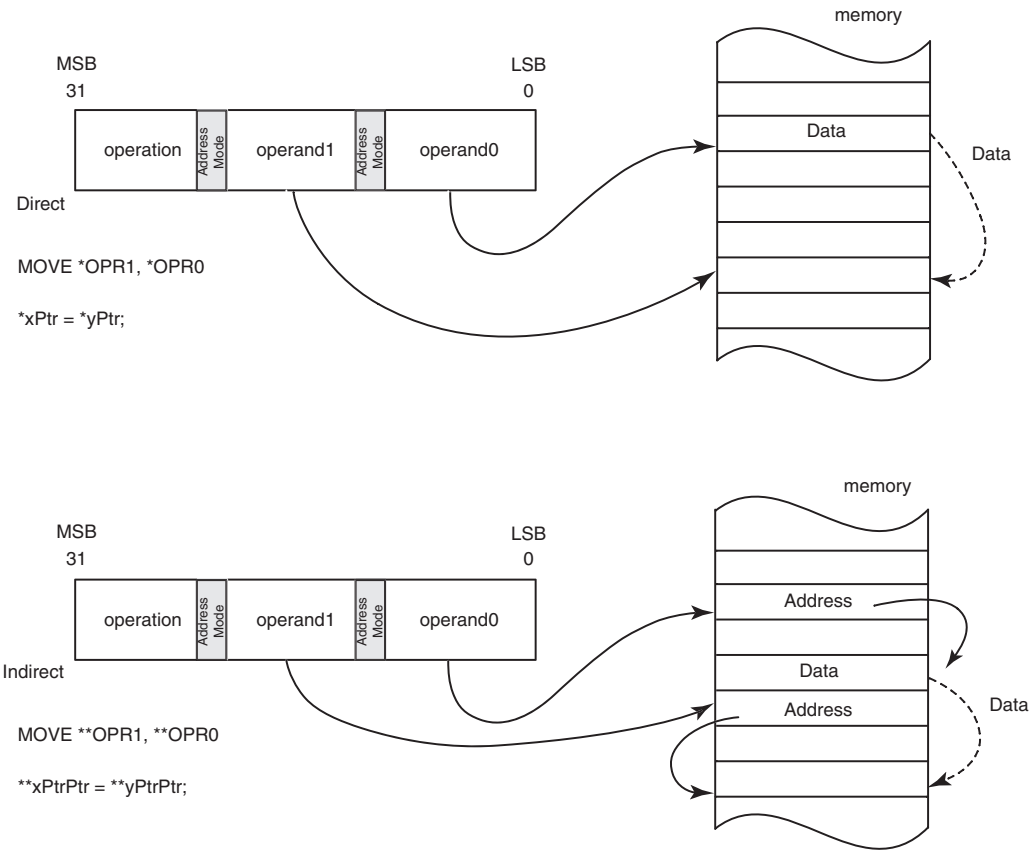

Figure 1.22 Register Direct and Register Indirect Instruction Formats

For the *indirect* operation, at the C/C++ level, the value of one variable, stored in memory and pointed to by the pointer variable *yPtrPtr,* is assigned to a second variable pointed to by a second pointer variable, *xPtrPtr*. At the assembly language level, the *MOVE* instruction now directs that the contents of one memory location serve as the address in memory of the operand that is to be assigned to the location in memory identified by the second operand.

indirect
yPtrPtr
xPtrPtr

The double ** symbols preceding the operands in the indirect access mode indicate that two levels of indirection are necessary to reach the final operand in memory.

The flow and two representative instructions at the assembly and the C and C++ levels are illustrated in Figure 1.22.

1.9.2.1.3 REGISTER DIRECT AND REGISTER INDIRECT MODES

register direct, register indirect

The distinction between the *register direct* and *register indirect* modes lies in the content of the referenced register. In the former case, the register contains the value of the operand and in the latter case the address (in memory) of the operand. The register indirect mode provides the means to easily implement pointer-type operations that are commonly used in C and C++.

The major disadvantage of indirect addressing is that an additional memory access is necessary to retrieve the operand's value. In contrast, when utilizing direct addressing, the value of the operand is found in the register.

register direct

In Figure 1.23, two different data transfer operations are shown. For the *register direct* operation, at the C/C++ level, the value of one variable, *y*, is assigned to a second variable, *x*. At the assembly language level, we assume that the values for *x* and *y* have previously been stored in registers R2 and R3, respectively. The *MOVE* instruction directs that the contents of R3 be copied to R2.

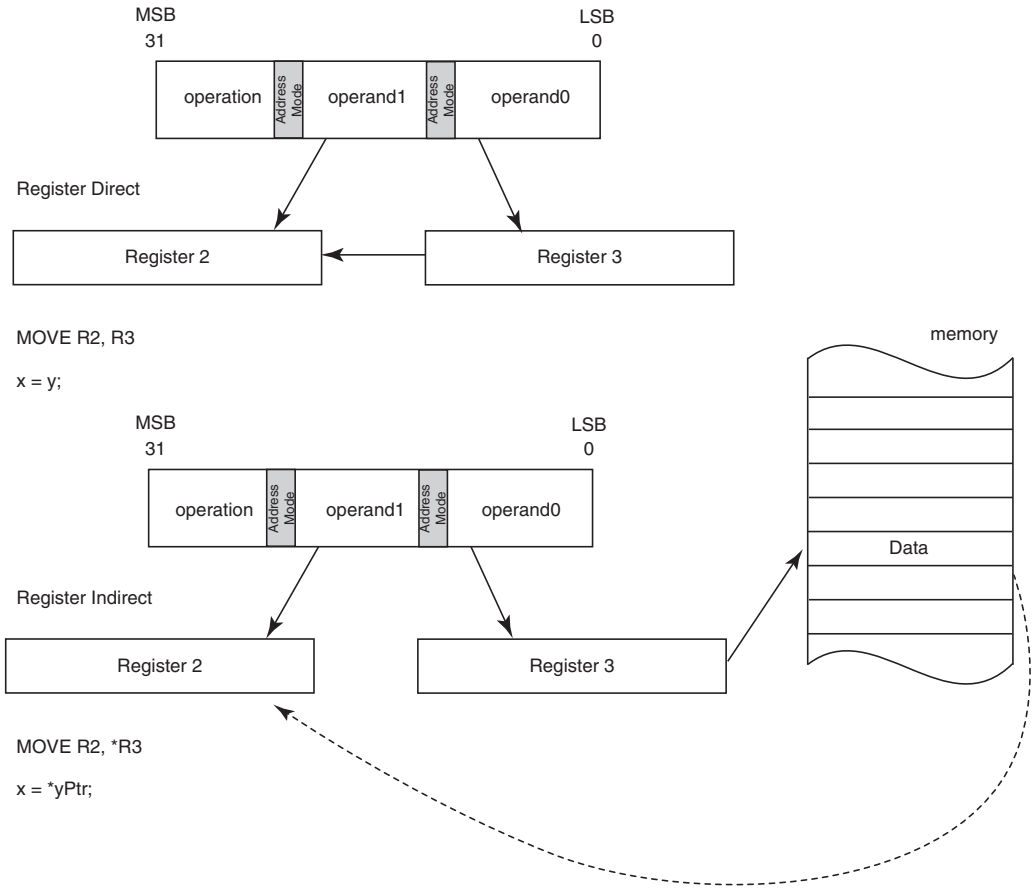

Figure 1.23 Register Direct and Register Indirect Data Transfer Operations

register indirect
yPtr, x

For the *register indirect* operation, at the C/C++ level, the value of one variable, stored in memory and pointed to by the pointer variable *yPtr,* is assigned to a second variable, *x*. At the assembly language level, once again we assume that the values for *x* and *yPtr* have been previously stored in registers R2 and R3, respectively. The *MOVE* instruction now directs that the contents of R3 serve as an address into memory; the value of the variable at that address is to be retrieved and to be copied into R2.

If the values and address modes of the two operands are interchanged, the data transfer would be from R2 into the location in memory pointed to by the contents of R3.

The flow and two representative instructions at the assembly and the C and C++ levels are illustrated in Figure 1.23.

The * preceding the second operand in the indirect instruction indicates that the assembler is to set the indirect addressing mode for the instruction.

1.9.2.1.4 INDEXED MODE

indexed, displacement

The *indexed* or *displacement* addressing mode provides support for accessing container-type data structures such as arrays. The effective address is computed as the sum of a base address and the contents of the indexing register. It is important to note here that following the execution of the instruction, neither the base address nor the index values are changed.

The major disadvantage of indexed addressing is the time burden associated with computing the address of the operand and then retrieving the value from memory. Indexing adds a greater burden to system performance than does indirect addressing.

Figure 1.24 illustrates the retrieval of an indexed variable using code fragments written at the assembly and the C or C++ levels.

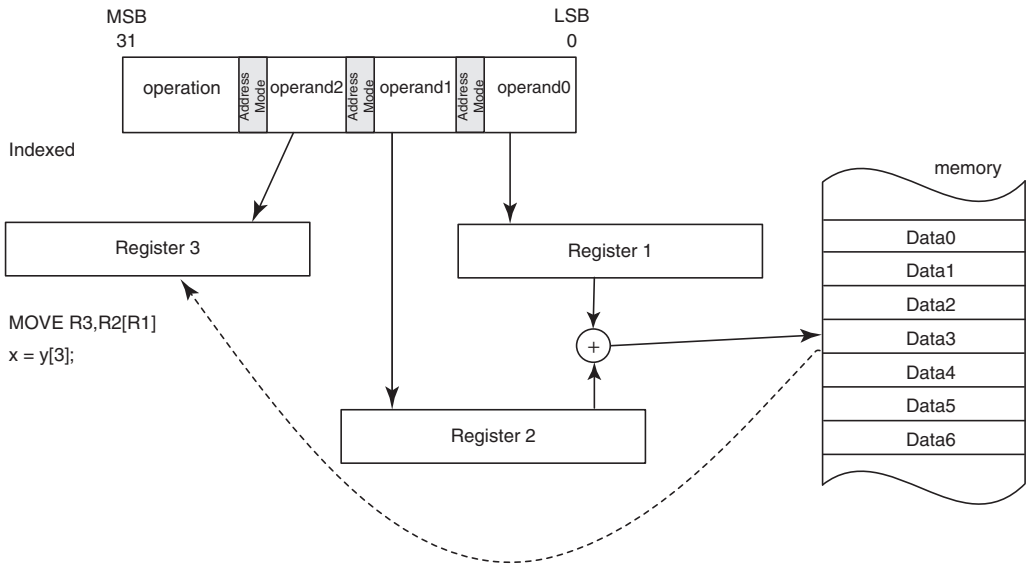

Figure 1.24 Indexed Mode Data Transfer Operations

Starting at the C/C++ level, we have an array variable named *y* and an integer variable *x*. The variable *x* is to be assigned the value contained in the fourth element of the array.

At the assembler level, the C/C++ fragment gets translated into a three-operand instruction. The base register, R2, will hold the starting address of the container in this case, the address of the 0th element of the array named *y*. The value of the variable *y* contains the address of the variable *Data0*, the start of the array. Register R1 will serve as the index register – that is, provide the offset. At the assembly level, we assume that the register R1 has already been initialized to the value 3, the offset into the container.

When the instruction is executed, the contents of R1 are added to the contents of R2, giving an address in memory. The value of the data stored in memory at the computed address is retrieved and written into register R3.

1.9.2.1.5 PROGRAM COUNTER RELATIVE MODE

program counter relative

Recall that the program counter contains the address in memory of the next instruction to be executed. That said, *program counter relative* addressing is mechanically almost identical to the indexed addressing mode. Nonetheless, there are several important differences. Firstly, the value in the program counter serves as the base address and, secondly, the program counter is assigned the value of the computed effective address; that is, the contents of the program counter are modified as a result of executing the instruction. Finally, the offset that is added to the program counter is a signed number. Thus, the PC contents following the addition of the offset may refer to an address that is higher (the offset was positive) or lower (the offset was negative) than the original value.

Figure 1.25 illustrates the flow of the instruction. For this instruction, operand0 is serving as the index register and is holding a value that has already been stored in it. The effective address is computed by adding the contents of the register identified by operand0 (R1 in this case) to the contents of the program counter. The program counter contents are then updated to the new value and now refer to the instruction at the computed address.

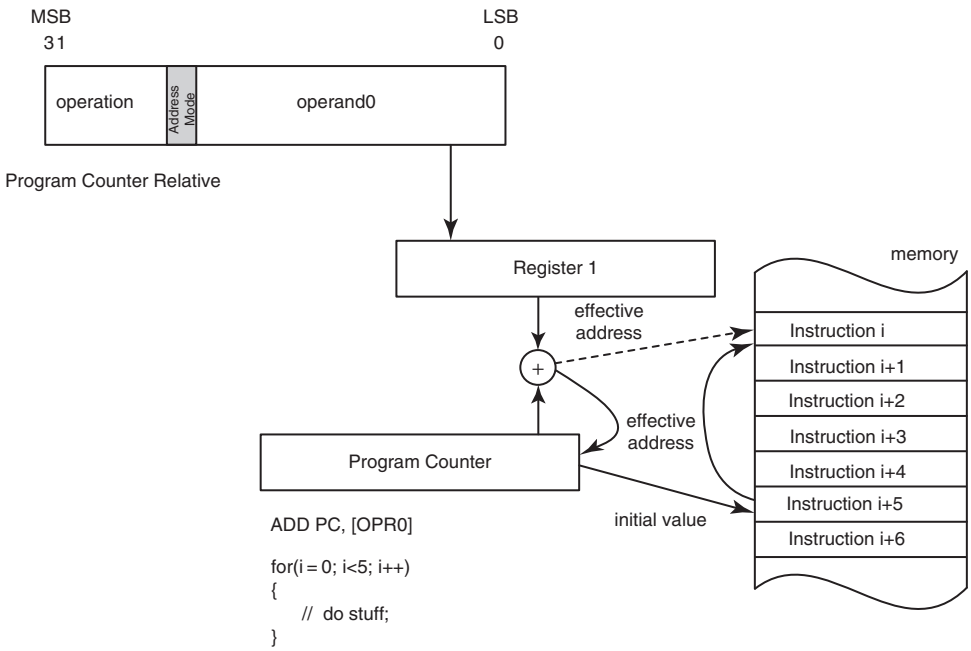

Figure 1.25 Program Counter Relative Operations

The C/C++ code fragment illustrates a simple *for* loop. Following the execution of the body of the loop, the flow must move back to the top of the loop and test the loop variable once again. A negative offset would have to be added to the contents of the PC to effect that movement.

The disadvantages of the PC relative mode are similar to those found in the indexed mode. There can be potential degradation of system performance.

1.9.3 Execution Flow

The execution flow or control flow captures the order of evaluation of each instruction comprising the firmware in an embedded application. We identify these as

- Sequential
- Branch
- Loop
- Procedure or Function Call

and will now examine each in turn.

1.9.3.1 Sequential Flow

Sequential control flow describes the fundamental movement through a program. Each instruction contained in the program is executed in sequence, one after another. A significant amount of the total code in an application is evaluated and executed in sequential order, although the individual sequences may be rather short. We capture that notion in the accompanying diagram in Figure 1.26, in the following C/C++ code fragment in Figure 1.27, and in assembler code in Figure 1.28.

The execution first assigns values to several variables and then performs an arithmetic operation on the two variables.

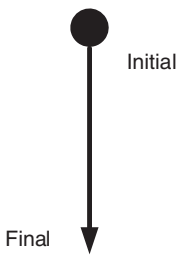

Figure 1.26 Sequential Flow

```
a = 10;
b = 20;
c = a + b;
```

Figure 1.27 C/C++ Sequential Flow

```
MOVE R1, #AH;       // puts 10 – hex A – into R1
MOVE R2, #14H;      // puts 20 – hex 14 – into R2
ADD R3, R1, R2;     // computes R1 + R2 and puts result into R3
```

Figure 1.28 Assembler Sequential Flow

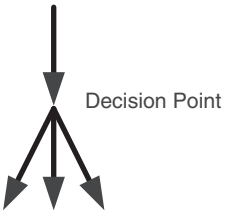

Figure 1.29 The Branch Construct

if else, switch, case

flag register
condition code register

1.9.3.2 Branch

A branching construct terminates a sequential flow of control with a decision point. At such a point, one of several alternate paths for continued execution is taken based on the outcome of a test on some condition. Graphically, such a construct is seen in Figure 1.29. The branch construct is used to implement an *if else, switch,* or *case* statement.

The branch may be executed unconditionally, in which case the contents of the PC are replaced by the effective address specified by the operand. Alternately, the branch may be taken conditionally based on the side effects of operations performed on data or on several different kinds of comparisons between two variables such as equality, a greater than or less than relationship, a carry from an arithmetic operation, or a variable being equal to or not equal to zero.

The conditional information is temporarily held as a collection of bits in a *flag register* or *condition code register*. The state of each bit in the register is evaluated and potentially changed following the execution of every instruction. Figure 1.30 lists some of the possible conditions that may be supported in a microprocessor.

E, NE	Operand1 is *equal/not equal to* Operand2.
Z, NZ	The result of the operation is *zero/not zero.*
GT, GE	Operand1 is *greater than/greater than or equal* to Operand2.
LT, LE	Operand1 is *less than/less than or equal* to Operand2.
V	The operation resulted in an *overflow*—the result is larger than can be held in the destination.
C, NC	The operation produced a *carry/no carry.*
N	The result of the operation is *negative.*

Figure 1.30 Typical Condition Codes

Branching alternatives that may be supported in a particular microprocessor are given in Figure 1.31.

BR label	unconditional branch to the specified label
BE label, BNE label	branch to the specified label if the equal flag is set or not set
BZ label, BNZ label	branch to the specified label if the zero flag is set or not set
BGT label	branch to the specified label if the greater than flag is set
BV label	branch to the specified label if the overflow flag is set
BC label, BNC label	branch to the specified label if the carry flag is set or not set
BN label	branch to the specified label if the negative flag is set

Figure 1.31 Typical Branching Instructions

The if-else construct is illustrated with the following C/C++ code fragment in Figure 1.32 and with assembler code fragments in Figure 1.33.

```
if (a == b)
        c = d + e;
else
        c = d – e;
```

Figure 1.32 C/C++ if-else Construct

```
CMP R2, R1          //  compare the contents of R1 and R2, will set the equal flag
BE $1               //  if the equal flag is set jump to $1
                    //  $1 is a label created by compiler
SUB R3, R4, R5      //  compute d – e and put results in c
BR $2               //  $2 is label created by compiler
$1: ADD R3, R4, R5  //   computed + e and put results in c
$2: ...
```

Figure 1.33 Assembler if-else Construct

1.9.3.3 If-else Construct

In the C code fragment in Figure 1.32, the two variables are compared. If they are equal, one arithmetic operation is performed; otherwise a second one is executed.

The code fragment in Figure 1.33 illustrates the construct in assembler. We assume that the variables a–e have been placed into registers R1–R5.

The compiler will create labels $1 and $2 if the original source was written in a high-level language or by the designer if the original source was assembler code.

1.9.3.4 Loop

The loop construct permits the designer to repeatedly execute a set of instructions either forever or until some condition is met. As Figure 1.34 illustrates, the decision to evaluate the body of the loop can be made before the loop is entered (*entry condition* loop) or after the body of the loop is evaluated (*exit condition* loop). In the former case, the code may not be executed, whereas in the latter, the code is executed at least once. The loop type of construct is seen in the *do, repeat, while,* or *for* statements.

entry condition
exit condition
do, repeat, while, for

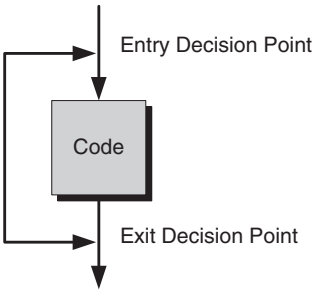

Figure 1.34 The Looping Construct

The C/C++ and assembler code fragments in Figures 1.35 and 1.36, respectively, illustrate a *while* loop construct.

```
while (myVar < 10)
{
    index = index + 2;
    myVar++;
}
```

Figure 1.35 C/C++ Looping Construct

```
$1: CMP R2, #AH      // test if R2 < 10
BGE $2               // if R2 greater than or equal to 10 branch to $2
ADD R3, #2H          // compute index + 2 put result in index
ADD R2, #1H          // add 1 to myVar
BR $1                // continue looping
$2: ....
```

Figure 1.36 Assembler Looping Construct

The body of the loop is continually evaluated as long as the loop variable is less than a specified value. This code fragment implements an entry condition loop.

myVar, index Assume that the variables *myVar* and *index* have been placed in R2 and R3, -respectively.

1.9.3.5 Procedure or Function Call

The procedure or function invocation is the most complex of the flow of control constructs. It is not more difficult; it is simply more involved. Such an invocation requires that the control flow leaves the current context, executes a set of instructions, and then returns to the original context, as we see in Figure 1.37. Such a construct is seen for a procedure or subroutine call, an interrupt handler, or co-routine.

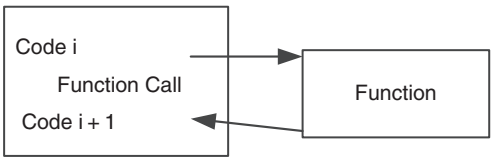

Figure 1.37 The Procedure Call

The operation is supported by the instructions given in Figure 1.38.

CALL operand	PC is unconditionally saved and replaced by specified operand; control is transferred to specified memory location.
RET	Previously saved contents of PC are restored, and control is returned to previous context.

Figure 1.38 Common Procedure Call Instructions

stack Before we examine the CALL process, we need to introduce a data structure called a *stack*.

1.9.3.5.1 STACK

The stack is a data structure that occupies an area in memory. It has finite size and supports several operations. Its structure is similar to an array except that, unlike an array, data can be entered or removed at only one location called the *top*. The top of the stack is equivalent to the 0th index in an array. When a new piece of data is entered, everything below is pushed down like a stack of trays in a cafeteria or like the last card in a discard pile in a card game. When a piece of data is removed, all data below move up, again, like a stack of plates or the new top card.

top

Figure 1.39 illustrates a model for the operations for several pieces of data. Data entry is called a *push* and data removal is called a *pop*. In reality, such a model is impractical because of the time burden in moving every piece of data each time a new entry is made. A more practical implementation adds or removes data at the open end of the structure. The memory address reflecting the current top of the stack is remembered and modified after each addition or removal. Such an address is called a *stack pointer*.

push, pop

stack pointer

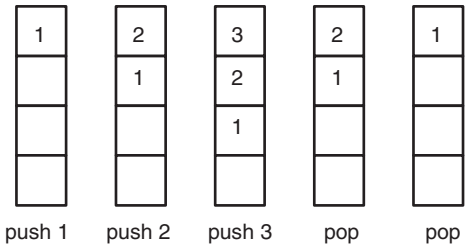

Figure 1.39 Stack Operations

Figure 1.40 presents a modified version of Figure 1.39 and illustrates how the stack pointer is properly managed.

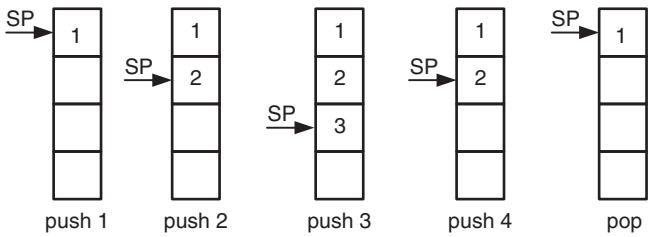

Figure 1.40 Managing the Stack Pointer

1.9.3.5.2 PUSH

The push operation puts something onto the top of the stack where it is held for later use. Mechanically, the push operation increments the address that is held by the stack pointer to refer to the next empty spot (the new top of the stack) and then writes the data to be stored into the address in memory designated by that address. As we see in Figure 1.40, for ease of implementation, the address contained in the stack pointer is typically incremented from a lower memory address to higher memory address.

1.9.3.5.3 POP

The pop operation takes something off the top of the stack by first retrieving the value in the memory location designated by the stack pointer and then decrementing the address that is

held by the stack pointer to refer to the next lower address (the new top of the stack). The retrieved value is returned as the result of the pop operation.

1.9.3.5.4 PROCESS

Let's now return to the function call. From a high-level point of view, code execution proceeds in a sequential manner until the function call is encountered. Flow of control switches to the function, the code comprising the function body is executed, and flow returns to the original context, as seen in Figure 1.41.

```
3000 Code
3053 CALL F1(3)
3054 pop R2
3055 More Code
....
5000 code              // Function Body....
5053 Return
```

Figure 1.41 Function Call Construct

Let's now examine the process in somewhat greater detail. In the code fragment shiwn in Figure 1.42, the program is initially loaded into memory and begins executing from address 3000. Code is executed until the flow reaches address 3053, at which point the function call is encountered. At this point, the sequence of operations shown in Figure 1.42 will occur.

1. The return address and parameters are pushed onto the stack.
   - The address saved is 3054.
   - The parameter saved is 3.
2. Address of function body 5000 is put into PC.
3. Instruction at 5000 begins executing.
4. Execution continues until 5053.
5. Return encountered
   - Stack gets
     - Return values
   - Stack loses
     - Return address
6. Return address is put into PC.
7. Flow returns to address 3054, and the top of stack is popped and put into register R2.
8. Execution continues at 3055.

Figure 1.42 Function Call Flow of Control

Had an additional function call been encountered in function F1, an identical process would have occurred. The process can be repeated multiple times; however, we must be aware that stack can overflow. If too much is pushed onto the stack, we begin to lose information, particularly the return address.

1.9.3.6 Arithmetic and Logic

Arithmetic and logical operations are essential elements in affecting what the processor is to do. Such operations are executed by any of several hardware components comprising the ALU. Figure 1.43 presents a block diagram for a possible functional ALU architecture.

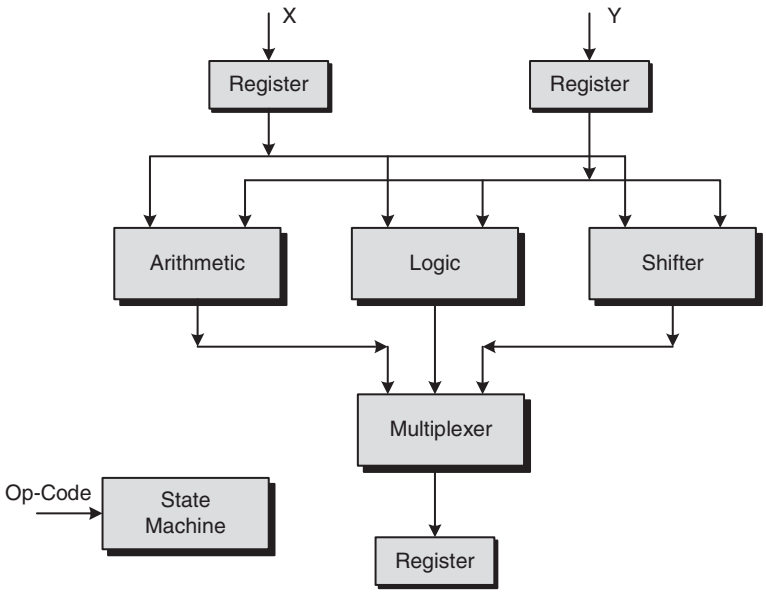

Figure 1.43 An ALU Block Diagram

Data are brought into the ALU and held in local registers. The op-code is decoded, the appropriate operation is performed on the selected operand(s), and the result is placed in another local register.

1.9.3.6.1 ARITHMETIC

add, subtract, multiply, divide

Typically, the processor will support the four basic arithmetic functions: *add, subtract, multiply,* and *divide*. Simpler processors will only implement the first two, relegating the last two to a software implementation by the designer. The add and subtraction operations may be supported in two versions, with and without carry and borrow.

The last two versions are intended to support double-precision operations. Such an operation is performed in two steps: the first computation holds any carry (borrow) and then utilizes that value as a carry in (borrow in) to the second step. Most such operations are implemented to support integer-only computations. If floating point mathematics is supported, a separate floating point hardware unit may be included. In addition to the four basic functions, the processor may also implement hardware increment and decrement operations.

At the instruction set level, a typical complement of arithmetic support will comprise the function listed in Figure 1.44.

ADD2, ADD3	// Two or three operands addition
ADDC	addition with carry
SUB2, SUB3	// Two or three operands subtraction
SUBB	subtraction with borrow
MUL	multiplication
DIV	division
INC	increment
DEC	decrement
TEST	operand tested and specified condition set
TESTSET	atomic test and set

Figure 1.44 Typical Arithmetic Instructions

1.9.3.6.2 LOGICAL OPERATIONS

Logical operations perform traditional binary operations on collections of bits or words. Such operations are particularly useful in embedded applications where bit manipulation is common. Such operations are discussed in detail in our studies of the software side of embedded systems. Typical operations included in the set of logical instructions are illustrated in Figure 1.45.

AND	bitwise AND
OR	bitwise OR
XOR	bitwise Exclusive OR
NOT or INV	complement
CLR or SET	clear or set
CLRC, SETC	carry manipulation

Figure 1.45 Typical Logical Instructions

1.9.3.6.3 SHIFT OPERATIONS

logical, arithmetic, rotate

Shift operations typically perform several different kinds of shifts on collections of bits or words. The major differences concern how the boundary values on either side of the shift are managed. Typically, three kinds of shift are supported: *logical, arithmetic,* and *rotate*. Any of the shifts may be implemented as a shift to the left or to the right.

A *logical* shift enters a 0 into the position emptied by the shift; the bit on the end is discarded. An *arithmetic* shift to the right propagates (and preserves) the sign bit into the vacated position; a shift to the left enters 0's on the right-hand side and overwrites the sign bit. The *rotate* shift circulates the end bit into the vacated bit position on the right- or left-hand side based on a shift to the left or to the right.

Typical operations in the set of shift instructions include those listed in Figure 1.46.

SHR operand, count	logical shift right
SHL operand, count	logical shift left
SHRA operand, count	arithmetic shift right
SHLA operand, count	arithmetic shift left
ROR operand, count	rotate right
ROL operand, count	rotate left

Figure 1.46 Typical Shift Instructions

1.10 EMBEDDED SYSTEMS – A REGISTER VIEW

transfer store, operate, make decisions

At the ISA level, the instruction set specifies the basic operations supported by the machine – that is, the external view of the processor from the software developer's perspective. During the early stages of design, it plays a significant role in the formulation of the architecture of the machine. The instruction set expresses the machine's ability to *transfer* data, *store* data, *operate* on data, and *make decisions,* all of which are necessary for the machine to be able to perform its ultimate task of aiding in solving problems.

control unit datapath

Underlying the instruction set is the physical hardware necessary to implement the operations directed by the instructions. The core hardware comprises a *control unit* and a *datapath* as illustrated in Figure 1.47.

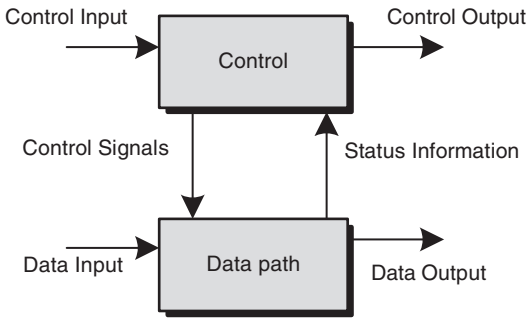

Figure 1.47 A Control and Datapath Block Diagram

microoperation

The datapath is a collection of registers and an associated set of *microoperations* on the data held in the registers. The control unit directs the ordered execution of the microoperations so as to effect the desired transformations of the data. Thus, the system's behavior (execution of the ISA level instructions) can be expressed by the movement of data among those registers, by operations and transformations performed on the register's contents, and by the management of how such movements and operations take place. The operations on data found at the instruction level are paralleled by a similar, yet more detailed, set of operations at the register level or RTL. When we study modeling of the hardware components of an embedded application, we will find that working initially at the RTL level is natural and convenient. Such an approach easily segues into the *hardware design language* (HDL) implementation.

register transfer level hardware design language (HDL)

1.10.1 The Basic Register

A register is a storage device that is capable of holding the collection of one or more bits. Based on the level of detail we need, we take several views of a register, as we see in the drawings in Figures 1.48 and 1.49.

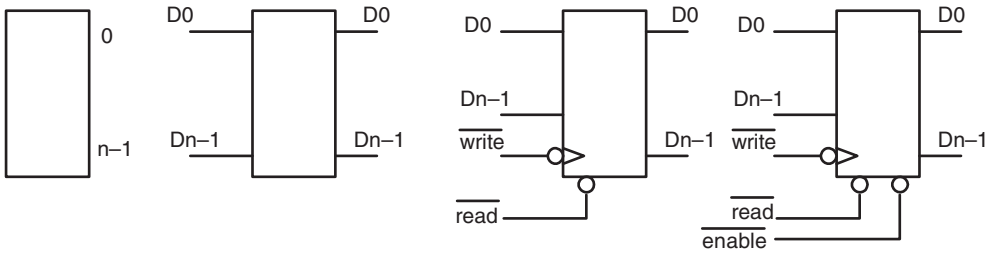

Figure 1.48 The Register at Several Levels of Abstraction – Parallel Data Entry

The abstract view on the left in Figure 1.48 shows a simple box with the bits numbered to reflect the size and outputs of the register. More refined/detailed views show inputs and outputs; those views may be further elaborated to include some control signals. Data are entered into the registers in parallel, as shown in Figure 1.48, or in serial, as illustrated in Figure 1.49.

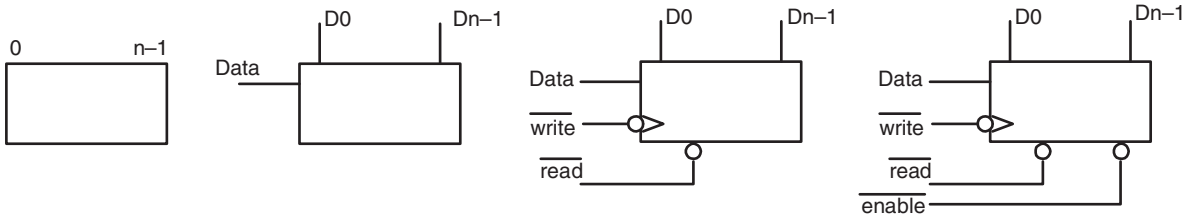

Figure 1.49 The Register at Several Levels of Abstraction – Serial Data Entry

1.10.2 Register Operations

Registers support two basic operations:

Read
Write

incrementing/ decrementing counting, shifting

These operations are illustrated in the timing diagrams shown in Figure 1.50; all other operations are built on these. Such higher level operations include *incrementing/ decrementing, counting,* or *shifting*.

1.10.2.1 Write to a Register

parallel write ***write*** ***serial write, write***

A *parallel write* operation begins when the data are placed onto the inputs of the register. Following a delay to allow the data to settle on the bus, the *write* signal is asserted. For a *serial write* operation, a *write* signal must accompany each data bit that is entered. In the drawings shown in Figure 1.50, the *write* signal is asserted low – which is typical.

Following each write operation, the contents of the register are changed to reflect the new values of the input data.

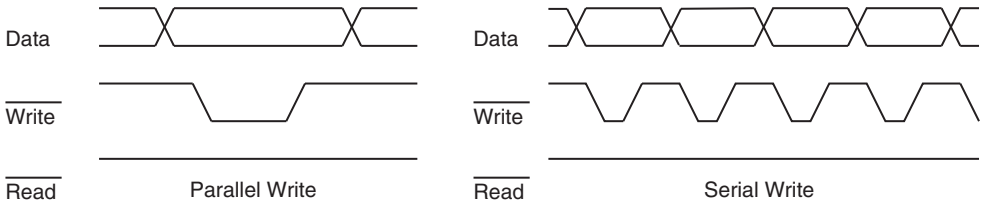

Figure 1.50 Writing to a Register

1.10.2.2 Read from a Register

read

The *read* operation is executed as shown in Figures 1.51. The *read* signal is issued; following some delay, the data appear on the register output. In this illustration, the read signal is shown as asserted low.

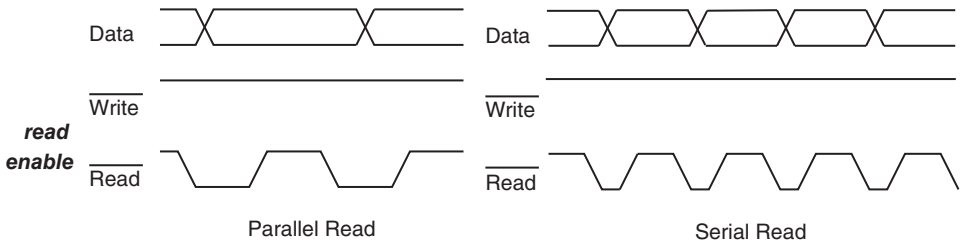

Figure 1.51 Reading from a Register

The output data will be a copy of the contents of the register; the state of the register is unchanged by the read operation.

read enable

In some designs, when the *read* signal is not asserted, the data output from the register is disabled; in others, output is disabled by an *enable* control signal.

1.11 REGISTER TRANSFER LANGUAGE

register transfer language ***register transfer notation (RTN)***

At the RTL, data transfers, operations on data, and control flow are described/specified using a *register transfer language*. Within the language, individual operations and transfers are expressed using *register transfer notation* (RTN). Table 1.3 summarizes the RTN notation that we will use in the remaining discussion. Such a notation has a direct equivalent in contemporary HDLs such as Verilog or VHDL, thereby facilitating the transformation from design to implementation.

Table 1.3 Register Transfer Notation

Operator	Operation
←	Transfer from right-hand side to left-hand side
→	If-then operation
[index]	Select word from memory at index
<index>	Select bit or bits from register at index or range
i..j	Index range
:=	Text substitution
#	Concatenation
:	Separator for parallel operations
;	Separator for sequential operations
@	Replication
{ }	Information about operation
( )	Grouping
= ≠ < ≤ > ≥	Comparison operators
+ − × ÷ +	Arithmetic operators
∧ ∨ ¬ ⊕ ≡	Logical operators

Table 1.4 illustrates how representative operations from the ISA level can be expressed using RTN.

Table 1.4 Instruction Set Architecture Operations Expressed in Register Transfer Notation

Type	Instruction	ISA level	Register transfer level
Data transfer	Move register	MOVE R1, R2	R1 ← R2
	Move from memory	MOVE R1, memadx	R1 ← (memadx)
	Move to memory	MOVE memadx, R1	(memadx) ← R1
	Move immediate	MOVE R1, #DEAD	R1 ← #DEAD
Control flow	Unconditional branch	BR $1	PC ← $1
	Conditional branch	BNE $1	cond (PC ← $1) if(cond) PC ← $1
Logic	Complement accumulator	CMA	A ← ¬A
	AND register	AND R1, R2	R1 ← R1 ∧ R2
	OR register	OR R1, R2	R1 ← R1 ∨ R2
	Shift register	SHL R1, #3	R1 <31..0> ← R1<31–n..0>#(n@0) Contents of R1 get replaced by bits in range of 31–n..0, where n is number of bits to shift and n 0s get extended on right
Arithmetic	ADD register with carry	ADDC R1, R2	R1 ← R1 + R2 + C
	Clear carry	CLRC	C ← 0
Program control	Don't execute an instruction	NOP	PC ← (PC + 1)
	Stop executing instructions	HALT	PC ← PC

1.12 REGISTER VIEW OF A MICROPROCESSOR

We will now examine the datapath and control for a simple microprocessor at the RTL level. We will begin by looking at the components that comprise the datapath from a register point of view. From there, we will look at the control of such a datapath by studying the instruction cycle for such a machine.

1.12.1 The Datapath

Figure 1.52 expresses the architecture of the datapath and the memory interface for a simple microprocessor at the RTL.

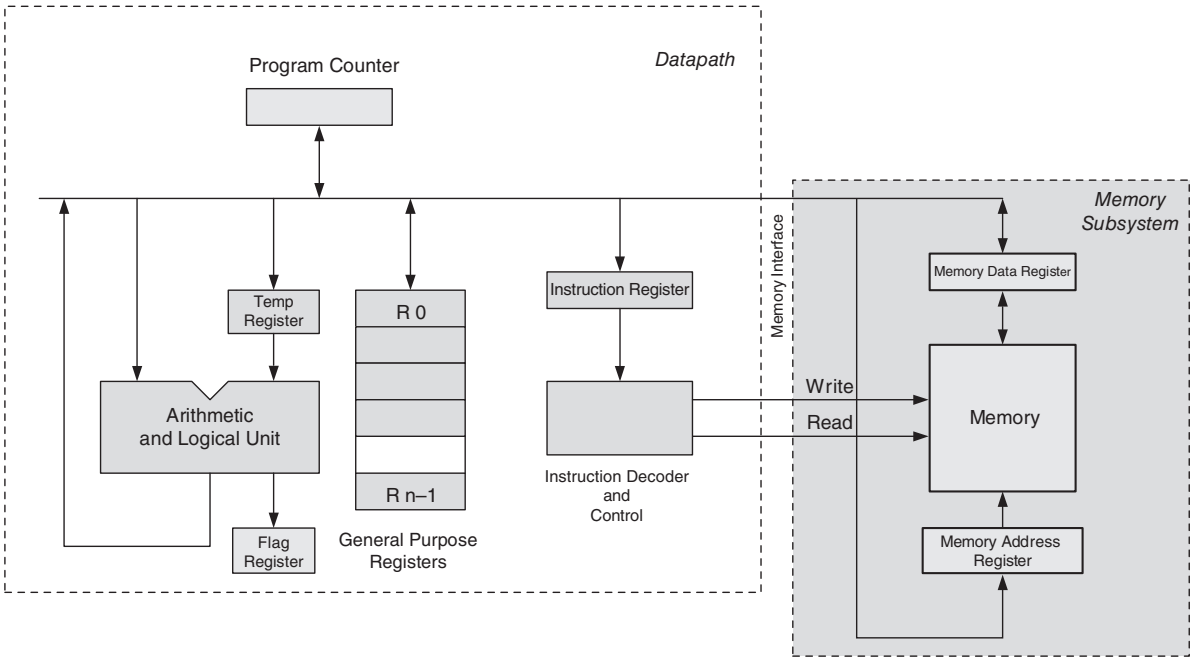

Figure 1.52 RTN Model for a Microprocessor Datapath and Memory Interface

In the diagram, we can identify the minimal set of registers, as listed in Figure 1.53.

Program Counter – PC	Hold next instruction address
Instruction Register – IR	Hold current instruction
General Purpose Registers – R0–Rn–1	Temporary data store
Memory Address Register – MAR	Hold address during read or write operation
Memory Data Register – MDR	Hold address during read or write operation
TR0	Hold operand during ALU operation
TR1	Hold the result of an arithmetic operation

Figure 1.53 Typical Microprocessor Register

1.12.2 Processor Control

instruction cycle

The control of the microprocessor datapath comprises four fundamental operations defined as the *instruction cycle*. These steps are identified in Figure 1.54, and are further described according to state diagram in Figure 1.55.

Fetch	Fetch instruction
Decode	Decode current instruction
Execute	Execute current instruction
Next	Compute address of next instruction

Figure 1.54 Steps in the Instruction Cycle

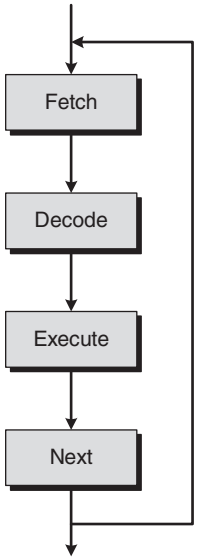

Figure 1.55 The Instruction Cycle

1.12.2.1 Fetch

fetch

The *fetch* operation retrieves an instruction from memory. That instruction is identified by its address, which is the contents of the program counter, PC. Thus, at the ISA level, the *fetch* operation is written as

```
MOVE IR, *PC;
```

Move the memory word identified by the address contained in the program counter into the instruction register (IR).

fetch
Memory Address Register*, *Read Memory Data Register
Instruction Register, fetch

The first step in the *fetch* operation places the contents of the program counter (which identifies the address of the next instruction) into the *Memory Address Register* (MAR). A *Read* command is issued to the memory, which retrieves the instruction stored in the addressed location and places it into the *Memory Data Register* (MDR). The contents of the MDR are then transferred to the IR. At the RTL level, the *fetch* decomposes into the sequence of steps given in Figure 1.56.

```
MAR ← PC;              // PC enabled out to bus, MAR captures value
MDR ← Memory[MAR];     // contents of specified memory location placed into MDR
IR ← MDR;              // MDR enabled out to bus, IR captures value
```

Figure 1.56 Components of the Fetch Instruction

READ
read

The second step in this sequence executes a *READ* operation from the specified memory location. The underlying hardware will generate the *read* control signal and manage the underlying timing.

1.12.2.2 Decode

decode
Instruction Decoder
control logic, execute

The *decode* step is performed when the op-code field in the instruction is extracted from the instruction and decoded by the *Instruction Decoder*. That information is forwarded to the *control logic*, which will initiate the *execute* portion of the instruction cycle.

1.12.2.3 Execute

control logic
execute

Based on the value contained in the op-code field, the *control logic* performs the sequence of steps necessary to *execute* the instruction. Two examples are given in Figures 1.57 and 1.58.

```
// C Instruction
      *xPtr = y;
// ISA Level Instruction
      ST *R1, R2;
// RTL Level Instructions
      MAR ← R1;                 // R1 enabled out to bus, MAR captures value
      MDR ← Memory[MAR];        // contents of specified memory location placed into MDR
      R2 ← MDR;                 // MDR enabled out to bus, R2 captures value
```

Figure 1.57 An Execute Sequence

```
// C Instruction
      *zPtr = x + *yPtr;
// ISA Level Instruction
      ADD *R3, R1, *R2;
// RTL Level Instructions
// Assume that R2 and R3 already contain the desired addresses in memory
      TR0 ← R1;                 // R1 enabled out to bus, TR0 captures value
      MAR ← R2;                 // R2 enabled out to bus, MAR captures value
      MDR ← Memory[MAR];        // contents of specified memory location placed into MDR
      TR1 ← TR0 + MDR;          // MDR enabled out to bus, ALU adds TR0 and MDR
                                // places result in TR1
      MAR ← R3;                 // R3 enabled out to bus, MAR captures value
      MDR ← TR1;                // TR1 enabled out to bus, MDR captures value
      Memory[MAR] ← MDR;        // contents of MDR placed into specified memory location
```

Figure 1.58 An Execute Sequence

Store the contents of a register in a named location in memory.

Add the contents of a register to a piece of data stored in memory and place the result back into memory, but at a different location.

1.12.2.4 Next

The address of the next instruction to be executed is dependent on the type of instruction to be executed and, potentially, on the state of the condition flags as modified by the recently completed instruction. At the end of the day, most reduce to algebraically adding a value to the PC. For short jumps, the displacement may be contained in one of the operand fields of the instruction; for longer jumps, the value may be contained in the memory location following the instruction.

next Thus, at the ISA level, the several versions of the *next* operation are written as

```
ADD PC, offset;
```

Algebraically modifying the PC is best accomplished by using one of the arithmetic functions in the ALU. The operation begins when the control logic places the desired offset into the ALU's temporary register. Next, the output of the PC is directed to the other ALU input. The ADD instruction is executed and the result is entered into the PC. Placing a specific value into the PC can be done directly by the control logic, as the target address is generally contained in the instruction.

next At the RTL level, the *next* step decomposes into the sequence of steps given in Figure 1.59.

```
// Assume the offset is contained in the instruction
TR0 ← IR<n..m>;        // offset field of instruction enabled out to bus, TR0 captures value
TR1 ← TR0 + PC;        // PC enabled out to bus, ALU adds TR0 and PC
PC ← TR1               // TR1 enabled out to bus, PC captures value
```

Figure 1.59 The Next Sequence

The Verilog program in Figure 1.60 implements a behavioral model of a portion of the datapath and control for the simple CPU presented at the start of this section. The number of registers has been reduced, only two instructions are implemented, and the address mode field supports four different modes. Nonetheless, the architecture implements a working system. Included are the test bench and the tester for the CPU.

```
#define TRUE            1'b1
#define FALSE           1'b0
/*
 instruction format 32 bit word
        31..24          op-code
        23..22          address mode field operand 1
        21..12          operand 1
        11..10          address mode field operand 0
        9..0            operand 0
 all registers are 32 bits
*/

// Build a test bench to test the design
module testBench;

wire [31:0]             pc;                                     // connect the pc
wire [31:0]             ir;                                     // connect the ir
wire                    clock;                                  // connect the clock
hal0                    aComputer (pc,ir, clock);               // build an instance of the computer
testIt                  aTester(clock, pc, ir);                 // build a tester

endmodule

// Test module
module testIt(clock, pc, ir);
// declare the input and output variables
input [31:0]            pc;                                     // program counter
input [31:0]            ir;                                     // instruction register
output                  clock;                                  // system clock
reg                     clock;                                  // system clock

parameter halfPeriod = 1;

initial
clock = 0;

// manage the clock
always
begin
#(halfPeriod) clock = ~clock;
end

// manage the display and look for changes
always @(posedge clock)
begin
        $monitor ($time,, "pc = %h \t ir = %h", pc, ir);        // record only changes
        #(10*halfPeriod);                                       // let clock cycle a couple of times
        #(halfPeriod);                                          // needed to see END of a simulation
        $stop;                                                  // stop so user can look at waveform
        $finish;                                                // exit
        end
endmodule
```

Figure 1.60 Model of the Datapath and Control for a Simple CPU

```
// The Computer - Hal0
module hal0 (pc, ir, clock);

// declare the I/O and registers
input           clock;
output [31:0]   pc;
output [31:0]   ir;

reg [31:0]      m [0:15];       // 16 x 32 bit memory
reg [31:0]      pc;             // 32 bit program counter
reg [31:0]      acc;            // 32 bit accumulator
reg [31:0]      ir;             // 32 bit instruction register
reg [31:0]      r[0:7];         // 8 32 bit general purpose registers

reg             notDone;        // flag to end program

integer         i;

// define op-codes

parameter add   =  8'h01;       // 8 bit add op-code
parameter move  =  8'h05;       // 8 bit move op-code
parameter done  =  8'hff;       // 8 bit done op-code

// define address mode field values

parameter dir   =  2'b00;
parameter ind   =  2'b01;
parameter imm   =  2'b10;
parameter pcr   =  2'b11;

// define registers

parameter r0    =  32'h0;
parameter r1    =  32'h1;
parameter r2    =  32'h2;
parameter r3    =  32'h3;
parameter r4    =  32'h4;
parameter r5    =  32'h5;
parameter r6    =  32'h6;
parameter r7    =  32'h7;

// initialize the system
    initial                         // initialize the pc and the accumulator
        begin
            pc = 0;                 // pc <- 0
            acc = 0;                // acc <- 0
            notDone = `TRUE;        // initialize notDone flag
    // define the instruction rom
    // enter some instructions into memory
            m[0] = 'h05000803;      // r0 <- 0x3
            m[1] = 'h05001802;      // r1 <- 0x2
            m[2] = 'h01001000;      // r1 <- r1 + r0
            m[3] = 'hFFFFFFFF;      // done - end of program
            m[4] = 'h00000000;
```

Figure 1.60 (*Continued*)

```
            m[5] = 'h00000000;
            m[6] = 'h00000000;
            m[7] = 'h00000000;

            ir = m [pc];                                    // fetch operation - get first instruction
end

   // run the program
         always
            while (notDone == `TRUE)
            begin
               @(posedge clock)                             // control system timing
               case(ir[31:24])                              // decode operation
                  move:                                     // move op-code
                  begin
                     $display("\nMove");                    // annotate execution

                     case(ir[11:10])                        // check address mode
                        dir:    $display("\ndirect");
                        ind:    $display("\nindirect");
                        imm:                                // implement immediate mode move
                        begin
                           $display("\nimmediate");         // execute operation
                           r[ (ir[21:12]) ] = ir[9:0];      // rx <- aValue

                           $display("\n register value %h", r[ (ir[21:12]) ]);
                        end

                        pcr: $display("\n pc relative");
                     endcase
                  end
                  add:                                      // add op-code
                  begin
                     $display("add\n");
                     case(ir[11:10])
                        dir:                                // register direct
                        begin
                           $display("\ndirect");           // execute operation

                           // rx <- rx + ry
                           r[ (ir[21:12]) ] = r[ (ir[21:12]) ] + r[ (ir[11:0]) ];
                           $display("\n register value %h", r[ (ir[21:12]) ]);
                        end

                        ind:   $display("\nindirect");
                        imm: $display("\nindirect");
                        pcr:   $display("\npc relative");
                     endcase
                  end
                  default:  $display("illegal op-code trap\n");   // identify illegal op-codes
               endcase
               pc = pc+1;                                   // next operation compute next address
               ir = m [pc];                                 // fetch operation
               if ( ir[31:24] == done )                     // check for end of program
                  notDone =`FALSE;                          // done
            end
```

Figure 1.60 (*Continued*)

1.13 SUMMARY

We introduced a high-level view at the computing core of an embedded application and then refined that core through several levels of increasing detail. We observed that such a core is usually manifest as a microprocessor, microcomputer, or microcontroller. We briefly examined each, identified the basic elements of each, and learned how they differ. We introduced and studied the architecture of the computing core beginning with the functional level, which comprises the four major blocks – input, output, memory, and CPU – as well as the bus as a means of interchanging information among the four blocks.

We then moved to the opposite end of the spectrum to learn how various essential kinds of information (numbers, characters, addresses, and instructions) are represented inside of a digital system. Building on the instructions, we learned that the instruction set drives or defines the architecture of the computer and how that architecture is refined and subsequently expressed at the register level. We designated these two levels as the ISA level and RTL of the computer.

At the ISA level, we examined how data and information are expressed within the machine, the different instruction formats, and the different addressing modes commonly supported by contemporary computing engines.

At the RTL level, we introduced the register, basic register operations, and the RTN used for expressing operations at the RTL. We then decomposed the CPU into the control and datapath components. We introduced the instruction cycle and examined how its fetch, decode, execute, and next constituents can be expressed at both the ISA and RTL levels.

We concluded with a behavioral level Verilog implementation of a simple datapath.

1.14 REVIEW QUESTIONS

The Hardware Side

1.1 Beginning with the computing core and moving to the complete system and its environment, the chapter identified a hierarchy of views of an embedded system. Please identify and briefly describe each of these views.

1.2 Identify and briefly describe the major functional blocks that comprise the computing core.

1.3 How are the major blocks of the computing core interconnected?

1.4 What are the major categories of signals flowing among the major blocks in the computing core?

1.5 What is meant by the term *bus width*?

1.6 Describe what is meant by the term *microprocessor*. Please be specific.

1.7 Describe what is meant by the term *microcomputer*. Please be specific.

1.8 Describe what is meant by the term *microcontroller*. Please be specific.

Representing Information

1.9 What kinds of information must we be able to represent in an embedded application?

1.10 What are the two basic classifications of numbers with which we are concerned in an embedded application?

1.11 How do we distinguish a signed integer from one that is unsigned?

1.12 What do the terms *little endian* and *big endian* mean? Why are they important?

1.13 When expressing a floating point number, what are the essential components that must be captured?

1.14 What do we mean when we say that the representation of a binary number has four bits of resolution?

1.15 A number that exceeds a microprocessor's word size may be truncated or rounded. Which of these will produce a greater error?

1.16 When arithmetic is performed on numbers with an error, the error will be reflected in the result of the calculation. How will that error affect the result if the numbers are added (subtracted), multiplied, or divided?

1.17 How are alphanumeric characters and symbols represented inside of a microprocessor?

1.18 How is an address expressed in a microprocessor word?

Instructions

1.19 An instruction is used to direct a microprocessor to perform some action. What are the major pieces of a microprocessor instruction?

1.20 What is meant by the *arity* of an instruction?

1.21 Please explain the terms *one-*, *two-*, or *three-operand instruction*.

1.22 Please explain the terms *one-*, *two-*, or *three-address instruction*.

1.23 What is the meaning of the term *op-code*?

1.24 What is the purpose of an *op-code*?

An Instruction Set View

1.25 Please explain the terms *instruction set* and *ISA*.

1.26 What is the purpose of the software tool called an *assembler*?

1.27 What is meant by the term *machine code*?

1.28 Microprocessor instructions can be classified into three major groups. What are these groups? Please describe what characterizes instructions in each of the groups.

1.29 Identify the hardware components that may be the source or destination of a data transfer instruction.

1.30 What information does the *addressing mode* of an instruction convey?

1.31 Please identify and briefly describe the more commonly used addressing modes discussed in the chapter.

1.32 How is the addressing mode information incorporated into an instruction?

1.33 Please explain the meaning of the term *execution* or *control flow*.

1.34 What are the four major categories of execution flow through an embedded program? Briefly describe what each means.

1.35 What is the purpose of a *flag* or *condition code register*?

1.36 What is the difference between an entry condition loop and an exit condition loop?

1.37 Please identify and describe the necessary steps for executing a function or procedure call. Please be precise.

1.38 What is a stack?

1.39 What are the major access operations that can be performed on a stack?

1.40 What is the purpose of the variable called the stack pointer?

1.41 What is the function of the ALU?

A Register View

1.42 Underlying the microprocessor's instruction set is the implementation hardware. What are the core components of that hardware?

1.43 What is meant by the expression RTL? How does the RTL view of a microprocessor relate to the ISA level view?

1.44 What is a register, and what is its purpose in a microprocessor?

1.45 What basic operations can be performed on a register?

1.46 What is the purpose of a register transfer language and RTN?

1.47 Please identify and briefly describe the registers that one will typically find in a microprocessor's datapath.

1.48 The control of a microprocessor's datapath is made up of four operations called the instruction cycle. Please identify and describe each of these operations.

1.15 THOUGHT QUESTIONS

The Hardware Side

1.1 Three kinds of computing engine are utilized in embedded systems. What are the advantages and disadvantages of each?

1.2 Under what circumstances should one consider using a microcontroller? microcomputer? microprocessor? Please explain your answer in detail.

1.3 Discuss the advantages and disadvantages of a wide versus narrow internal system bus.

1.4 Is it necessary for the address and data portions of the system bus to have the same number of bits? What are the pros and cons of a wider address bus? data bus?

1.5 In some designs, the address and data signals are multiplexed onto the same set of bus lines. What are the advantages and disadvantages of such a scheme?

1.6 Discuss the benefits gained from and the disadvantages of a von Neumann versus Aiken Machine.

Representing Information

1.7 What are the limitations on the amount of information that can be stored in a data word?

1.8 What are the trade-offs of a wide versus narrow word size?

1.9 If the system bus is 1 byte wide, how can a 32-bit word be transferred over the bus?

1.10 If an embedded system is designed around a 16-bit word, is it possible to support 32-bit floating point numbers? If so, how and if not, why not?

1.11 Identify several reasons for using unsigned integers.

1.12 If the word size in an embedded system is 16-bits, why would one ever use bytes? Unsigned bytes?

1.13 If an embedded core is implemented as a *big endian* machine, how can one communicate with a peripheral device that is *little endian*?

1.14 Why is type information necessary for data words?

1.15 The hidden bit format for floating point numbers provides an extra bit of resolution for free. Someone has suggested applying the same technique to gain two extra bits. What do you think of the idea? Are there any problems with it?

Instructions

1.16 The essential components of an instruction are the op-code and the operand(s) on which the operation is to be performed. Is it necessary that the op-code always contain the same number of bits? Why or Why not?

1.17 The chapter discussed one-, two-, and three-address instructions. Is it possible to have a zero-address instruction? What are the benefits of such a design?

1.18 How can double indirection be implemented in an instruction?

An Instruction Set View

1.19 What are the four major pieces of information that an instruction must convey? How are these done?

1.20 In the chapter, several different addressing modes were discussed. Can you think of others that might be useful?

1.21 Are the bits in the *machine code* representation of an instruction arbitrary, or do they have a specific meaning? If they have meaning, what might it be?

1.22 Microprocessor instructions can be classified into three major groups. What are these groups? Please describe what characterizes instructions in each of the groups.

1.23 If an instruction contains the address of an operand in memory, how can the source or destination of an operation be conveyed if the source or destination is an input or output port on the microprocessor?

1.24 If one has an assembly code listing for an embedded program that has been running on a Motorola processor, will that program run on an Advanced Micro Devices (AMD) processor?

1.25 If one has a C code listing for an embedded program that has been running on a Motorola processor, will that program run on an AMD processor?

1.26 Please explain how a branch type of instruction works from an instruction set point of view.

1.27 How does an instruction know how or where to access a *flag* or *condition code register*?

1.28 Discuss ways of managing the case of the instruction decoder finding an op-code in an instruction that it does not recognize.

A Register View

1.29 A RISC architecture frequently incorporates many more registers than does a CISC design? Why is this?

1.30 Explain why a register access is generally faster than a memory access.

1.31 Explain how an indexed type of instruction might be implemented at the register level.

1.32 How can data from multiple sources be transferred into the same register?

1.33 Discuss the advantages and disadvantages of building a register from latches versus flip-flops.

1.34 What are the advantages and disadvantages of tristate gates versus a traditional multiplexer gate for transferring data from one register to another?

1.16 PROBLEMS

1.1 Express the following decimal numbers in the bases indicated.

Decimal: 1011, 23.4, 207, 111.439

(a) Binary
(b) Octal
(c) Hexadecimal
(d) Binary Coded Decimal (BCD)

1.2 Express the following binary numbers in the bases indicated.

Binary: 101101011, 1101.11001, 1001001110, 111.001

(a) Decimal
(b) BCD
(c) Hexadecimal

1.3 Express the following hexadecimal numbers in the bases indicated.

Hexadecimal: B3D9, CA.43, 1234, 5D.06F

(a) Decimal
(b) BCD
(c) Binary

1.4 Express the following decimal numbers in the bases indicated.

Decimal: 12.34, 9503.313

(a) Binary
(b) Hexadecimal

1.5 Express your given name and your family name as ASCII characters in the following format:

givenName familyName

The ASCII characters should be written in their binary form and stored so as to use the smallest number of 16-bit words. Check the Web site http://asciitable.com for the set of ASCII characters.

1.6 We have the following data stored in the first 32 locations of a memory as shown.

Memory address	Value
00000	0000010
	0101001
	1000001
	0101111
	0001111
	0000011
	1100101
	1101001
	1111101
	0001111
	0011101
	0010111
	0001011
	0000001 0101111
	0101001
	0100011
	0000010
	0001101
	0000111
	0111101
	0111001
	1101001
	0011101
	0000011
	0001111
	0011111
	0010001
	0111101
	1001101
	1001101
11111	1010101

(a) Please identify the words at the following hexadecimal addresses.

12, F, 1E, 7, 1C

(b) Please give the hexadecimal addresses for the following words in memory.

1100101, 0010111, 0100011, 0111101

1.7 We have the following C code fragment:

```
unsigned char a = 0xD3;
char b = 'A';
int c = 6;
int d = 9;
int e = -31564;
unsigned int f = 0xFAD7;
float g = 3.1;
float h = 0.0345;
int j;
int* cPtr = &c;
int* dPtr = &d;
int** cPtrPtr = &cPtr;
int** dPtrPtr = &dPtr;
float* gPtr = &g;
float* hPtr = &h;
```

Variable	Memory address	Value
00000	1000	
	1001	
	2000	
	2001	
	3000	
	3001	
	4000	
	4001	

(a) Please put the values, in binary, for all of the variables into memory. Assume 16-bit words and big endian notation.

(b) How do the values in memory change after the following code fragment is executed?

```
e = e + f;
*gPtr = *gPtr + *hPtr;
j = **cPtrPtr - **dPtrPtr;
```

1.8 Convert each of the following floating point numbers into binary. Assume a 16-bit word for which the weight of the least significant bit is 2^{-4}. Following the conversion, what is the error for each number?

(a) E = 72.23

(b) F = 121.034

(c) G = 98.6

(d) H = 43.612

1.9 Using the numbers and errors from Problem 1.8, perform the following calculations. What is the worst case error for each calculation?

(a) E + F + G + H

(b) E*F

(c) G*H

(d) (G*H)/(E + F)

1.10 Please show the contents of the stack, the position of the stack pointer (SP), and the contents of the indicated registers after the execution of each of the following instructions. The instructions are executed in sequence.

Initially, the registers contain the following values `R0 = 1234, R1 = 2345, R2 = 4567, R3 = 8901.`

For two-operand instructions, assume that the left-hand operand is also the destination.

PC	Instruction
FACE	ADD R3, R1
FACF	PUSH R2
FAD1	POP R3
FAD6	ADD R1, R2

Registers			*Stack*
R0			
R1			
R2			
R3			
PC			

Registers			*Stack*
R0			
R1			
R2			
R3			
PC			

Registers			*Stack*
R0			
R1			
R2			
R3			
PC			

Registers			*Stack*
R0			
R1			
R2			
R3			
PC			

1.11 Using the assembly language instructions introduced in the chapter, write a program to solve the following problem.

$$\left(\frac{(A \cdot B)}{(C - D)}\right) \cdot E$$

The variables A–E are already stored in the registers R1–R5. The result is to be placed into register R6. You may not change what is stored in registers R1–R5.

1.12 How does your program in Problem 1.10 change if the registers contain the addresses of the variables rather than the values of the variables?

1.13 Given the memory locations, values below, and a one-address machine with an accumulator (the accumulator is

the default destination), what values do the following instructions load into the accumulator?

Assume that R1 and R3 contain the values 0×5000 and 0×3000, respectively.

Memory location	*Contents*
1000	2000
2000	3000
3000	FACE
4000	5000
5000	1000

Accumulator

```
MOVE R1 5000
LOADI 0x5000
MOVE *R1
LOADI 0xFACE
MOVE *R2 3000
```

1.14 A partial set of specifications for a typical embedded processor are given as:

– 16-bit architecture – 16-bit-wide data words.
– 20-bit instructions.
– 8 general-purpose registers.
– One-, two-, and three-address instructions.
– Each address field is qualified by a 2-bit field to identify any of the following four different addressing modes: *register direct, register indirect, pc relative, indexed*.

(a) Please give the format for a one-, two-, and three-address instruction for this machine. Observe that the width of the instruction is different from the width of the data. This is not a problem.

(b) Please show how the following instructions would be implemented using the format you designed.

```
ADD2, SUB2, MUL, DIV
opcodes: 00001, 00010, 00011, 00100
respectively
```

add, subtract, multiply, divide two operands.

```
MOVE      opcode: 00101
```

Move a word from memory into a register.

Move a word to a memory address location that is contained in a register.

```
JUM       opcode: 00110
```

Jump to a location.

```
BR, BE    opcodes: 00111, 01000
```

Load a 16-bit integer into a register.

```
CMP       opcode: 01001
```

Compare two operands and set the appropriate condition code.

```
e CALL, RET      opcodes: 01010, 01011
```

PC is replaced by a specified address and control is transferred to the address.

Control is returned to the calling context.

1.15 We have the following requirements for a microprocessor. Assume that the processor has 32-bit instructions. Let each register operand address be specified by a 5-bit field for the address and qualified by a 2-bit field to identify any of the following four different addressing modes: *register direct, register indirect, pc relative, indexed*.

Assume there are K two-operand instructions and L zero-operand instructions (e.g. HALT) required.

(a) What is the maximum number of one-address instructions that can be provided in the computer?

(b) Give the format for zero-, one-, and two-address instructions. How are these distinguished?

(c) How would the following instructions be expressed using your format:

- Move a word from memory into a register.
- Move a word to a memory address location that is contained in a register.
- Jump to a location.
- Add, subtract, multiply, divide two operands.
- Load the integer 59 into a register.
- Read from the keyboard.

1.16 Consider the following state of a computer:

```
Register R1 contains 800
Register R2 contains 3000
Memory location 1000 contains 2000
Memory location 2000 contains 3000
Memory location 3000 contains 1000.
```

All numbers are expressed in hex notation.

For the following three instructions, each is executed from the above initial state.

(a) What is the effect of executing each instruction?

(b) How many words does each instruction occupy?

(c) How many memory accesses does the fetching and execution of each instruction require?

Assume each instruction is independent of the others

```
MOVE *R2, R1        // move R1 to what R2 is
                    // pointing at
MOVE *R2,1800(R1)   // move the contents
                    // of the memory location
                    // indexed by R1 to what R2
                    // is pointing at
MOVI *R2, #DEAD     // move the constant
                    // DEAD to what R2
                    // is pointing at.
```

1.17 An embedded application executing on the microprocessor in Problem 1.15 is required to perform the following computation:

$$Z \neg W_1 g Y_1 + W_2 g Y_2 + W_3 g Y_3$$

on the contents of memory locations Z_1, W_1, Y_1, W_2, Y_2, W_3, and Y_3. None of the contents of locations W or Y may be destroyed.

(a) Write a straight-line program for the task described above.

(b) Write a loop program for the same task on the same computer.

(c) Calculate the number of main memory accesses (Read or Write operations) required for each program. Include all accesses required or fetching and executing the instructions. For example, the instruction:

```
MOVE Ri, Rj
```

requires two memory accesses: one for reading the instruction from main memory into the microprocessor and one for reading the operand from main memory location contained in R_i into the microprocessor.

1.18 In some computers, subroutine linkage is implemented in the following way. The call subroutine instruction stores the return address (the next instruction in the calling program) in the first location of the subroutine and then branches to the second location, where the execution of the subroutine begins.

(a) Define a suitable instruction for returning from the subroutine.

(b) How would you pass parameters between the calling program and the subroutine?

(c) Would the above linkage support nesting of subroutines? Why or why not?

(d) Would the above linkage support recursion? Why or why not?

1.19 Based on the RTL model of a microprocessor datapath given in Figure 1.52 and the RTL operations given in Table 1.4, please identify the necessary steps that comprise each phase of the instruction cycle in order to perform the following instruction on that microprocessor:

```
// add the contents of the value
// contained in the register R3 to
// the contents of the memory
// location pointed to by R2 and
// put the results into R4.
add R4, R3, *R2;
```

1.20 We have the following C code fragment that adds one to each element of an array.

```
int a[4] = {1, 2, 3, 4}; // declare an array
                         // of integers
int i ;
for (i = 0; i < 4; i++)
a[i] = a[i] + 1;         // add 1 to each
                         // element of the array
```

(a) Using the instructions you developed in Problem 1.15 translate the given code fragment into assembler.

(b) Using the RTL operations given in Table 1.4, translate the assembler to RTL level.

(c) Using the instructions you developed in Problem 1.15 and the assembler from part (a), translate the code fragment into machine code.

1.21 The architecture known as the Aiken or Harvard architecture uses a separate data memory and a separate instruction memory.

(a) Please give two advantages and two disadvantages of the Harvard architecture in comparison to the Princeton architecture. Be precise in your answer.

(b) For the following C++ code fragment. Which memory, the instruction or the data memory, should contain the values for *x* and *y*? Please explain why.

```
int x = 3;
int y = 4;
x = x + y;
```

(c) Using the following memory diagrams, please place the values for *x* and *y* into locations in the appropriate memory.**

	Instruction memory	
Memory location	*Memory address*	*Contents*

	Data memory	
Memory location	*Memory address*	*Contents*

(d) Are the following assembly code instructions one-, two-, or three-address instructions? Please explain your answer.

```
LOADI Ri, #constant   // load the constant
                      // into Ri
LOADI *Ri, #constant  // load the constant
                      // into what Ri is
                      // pointing to
ADD2 Ri, Rj           // add the contents of
                      // Ri to Rj and place
                      // the result in Ri
ADD2 *Ri, *Rj         // add the contents of
                      // what Ri is pointing to
                      // what Rj is pointing to
                      // and place the result in
                      // what Ri is pointing to
```

(e) Using the instructions above, complete the following code fragment to implement the addition operation in part (b).

LOADI		
LOADI		
ADD2		

(f) Using the following memory diagrams, please place the instructions you wrote in part (e) into locations in the appropriate memory.

	Instruction memory	
Memory location	*Memory address*	*Contents*

	Data memory	
Memory location	*Memory address*	*Contents*

(g) Please provide a format for the add instruction given above. Explain the purpose of each field in your implementation.

(h) Please identify all of the steps necessary to execute the add instruction in part (e) above on the Harvard machine. Refer to the specific parts of the architecture diagram in your descriptions.

1.22 A computer is designed that uses a set of special registers, RS0–RS3, rather than the stack to pass data into and out of a subroutine. The stack is only used to hold the return address. The stack pointer is held in the special register SP and the program counter in register PC.

The machine has four general-purpose registers R0–R3.

The machine has a status flag, equal, that is appropriately set or reset for each instruction execution as indicated below.

Write a routine exp. (num, pow) that will compute num^{pow} for such a machine and return that value.

(a) Please write the detailed level pseudocode for invoking such a routine assuming that it is called from some higher level function. Be certain to identify *all* the steps.

(b) Please write the sequence of assembly language steps necessary to implement such a sequence using the following instructions.

```
LOAD Ri, Rj    // Ri ← Rj, sets the equal
               // flag true if Ri contains 0
LOAD Ri, #val  // Ri ← #val, sets the
               // equal flag true if Ri
               // contains 0
BE @Ri         // jumps to address
               // contained in Ri if the
               // equal flag is true
BNE @Ri        // jumps to address
               // contained in Ri if the
               // equal flag is false
MUL Ri, Rj     // Ri ← Ri * Rj, sets the
               // equal flag true if Ri
               // contains 0
DIV Ri, Rj     // Ri ← Ri / Rj, sets the
               // equal flag true if Ri
               // contains 0
```

```
ADD2 Ri, Rj    // Ri ← Ri + Rj, sets the
               // equal flag true if Ri
               // contains 0
SUB2 Ri, Rj    // Ri ← Ri - Rj, sets the
               // equal flag true if Ri
               // contains 0
INC Ri         // Ri ← Ri + 1
DEC Ri         // Ri ← Ri - 1, sets the
               // equal flag true if Ri
               // contains 0
PUSH Ri        // stack ← Ri
POP Ri         // Ri ← stack
```

(c) Using a value of 20 for num and 3 for pow, please indicate the contents of each of the following registers at the execution points indicated assuming the sequence of steps you have identified was executed. Also indicate the top of the stack.

Assume the original function call is at memory address 3050 and that the subroutine is at memory address 5000.

Prior to Function Call

PC	

GP Registers	
R0	
R1	
R2	
R3	

SP Registers	
RS0	
RS1	
RS2	
RS3	

Stack	

After the Function Call

PC	

GP Registers	
R0	
R1	
R2	
R3	

SP Registers	
RS0	
RS1	
RS2	
RS3	

Stack	

Prior to the Return

PC	

GP Registers	
R0	
R1	
R2	
R3	

SP Registers	
RS0	
RS1	
RS2	
RS3	

Stack	

After the Return

PC	

GP Registers	
R0	
R1	
R2	
R3	

SP Registers	
RS0	
RS1	
RS2	
RS3	

Stack	