UNIT 1

Embedded Computing

1.1 COMPLEX SYSTEMS AND MICROPROCESSORS

What is an ***embedded computer system***? Loosely defined, it is any device that includes a programmable computer but is not itself intended to be a general-purpose computer. Thus, a PC is not itself an embedded computing system, although PCs are often used to build embedded computing systems. But a fax machine or a clock built from a microprocessor is an embedded computing system.

1.1.1 Embedding Computers

A microprocessor is a single-chip CPU. Very large scale integration (VLSI) stet—the acronym is the name technology has allowed us to put a complete CPU on a single chip since 1970s, but those CPUs were very simple. The first microprocessor- the Intel 4004, was designed for an embedded application, namely, a calculator. The calculator was not a general-purpose computer—it merely provided basic arithmetic functions. However, Ted Hoff of Intel realized that a general-purpose computer programmed properly could implement the required function, and that the computer-on-a-chip could then be reprogrammed for use in other products as well. Since integrated circuit design was (and still is) an expensive and time- consuming process, the ability to reuse the hardware design by changing the software was a key breakthrough. The HP-35 was the first handheld calculator to perform transcendental functions [Whi72]. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor. How- ever, the ability to write programs to perform math rather than having to design digital circuits to perform operations like trigonometric functions was critical to the successful design of the calculator.

1.1.2 Characteristics of Embedded Computing Applications

Embedded computing is in many ways much more demanding than the sort of programs that you may have written for PCs or workstations. Functionality is important in both general-purpose computing and embedded computing, but embedded applications must meet many other constraints as well.

- *Complex algorithms:* The operations performed by the microprocessor may be very sophisticated. For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to opti- mize the performance of the car while minimizing pollution and fuel utilization.

- *User interface:* Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options. The moving maps in Global Positioning System (GPS) navigation are good examples of sophisticated user interfaces.

- *Real time:* Many embedded computing systems have to perform in real time— if the data is not ready by a certain deadline, the system breaks. In some cases, failure to meet a deadline is unsafe and can even endanger lives. In other cases, missing a deadline does not create safety problems but does create unhappy customers—missed deadlines in printers, for example, can result in scrambled pages.

- *Multirate:* Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time. They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of ***multirate*** behavior. The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized. Failure to meet a deadline on either the audio or video portions spoils the perception of the entire presentation.

- *Manufacturing cost:* The total cost of building the system is very important in many cases. Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required, and the types of I/O devices.

- *Power and energy:* Power consumption directly affects the cost of the hardware, since a larger power supply may be necessary. Energy con- sumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

1.1.3 Why Use Microprocessors?

There are many ways to design a digital system: custom logic, field-programmable gate arrays (FPGAs), and so on. Why use microprocessors? There are two answers:

- Microprocessors are a very efficient way to implement digital systems.

- Microprocessors make it easier to design families of products that can be built to provide various feature sets at different price points and can be extended to provide new features to keep up with rapidly changing markets.

1.1.4 The Physics of Software

Computing is a physical act. Although PCs have trained us to think about computers as purveyors of abstract information, those computers in fact do their work by moving electrons and doing work. This is the fundamental reason why programs take time to finish, why they consume energy, etc.

A prime subject of this book is what we might think of as the ***physics of software***. Software performance and energy consumption are very important prop- erties when we are connecting our embedded computers to the real world. We need to understand the sources of performance and power consumption if we are to be able to design programs that meet our application's goals. Luckily, we don't have to optimize our programs by pushing around electrons. In many cases, we can make very high-level decisions about the structure of our programs to greatly improve their real-time performance and power consumption. As much as possible, we want to make computing abstractions work for us as we work on the physics of our software systems.

1.1.5 Challenges in Embedded Computing System Design

External constraints are one important source of difficulty in embedded system design. Let's consider some important problems that must be taken into account in embedded system design.

How much hardware do we need?

We have a great deal of control over the amount of computing power we apply to our problem. We cannot only select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more. Since we often must meet both performance deadlines and manufacturing cost constraints, the choice of hardware is important—too little hardware and the system fails to meet its deadlines, too much hardware and it becomes too expensive.

How do we meet deadlines?

The brute force way of meeting a deadline is to speed up the hardware so that the program runs faster. Of course, that makes the system more expensive. It is also entirely possible that increasing the CPU clock rate may not make enough difference to execution time, since the program's speed may be limited by the memory system.

How do we minimize power consumption?

In battery-powered applications, power consumption is extremely important. Even in nonbattery applications, excessive power consumption can increase heat dis- sipation. One way to make a digital system consume less power is to make it un more slowly, but naively slowing down the system can obviously lead to missed deadlines. Careful design is required to slow down the noncritical parts of the machine for power consumption while still meeting necessary performance goals.

How do we design for upgradability?

The hardware platform may be used over several product generations, or for several different versions of a product in the same generation, with few or no changes. However, we want to be able to add features by changing software. How can we design a machine that will provide the required performance for software that we haven't yet written?

How Does it Really work ?

Reliability is always important when selling products—customers rightly expect that products they buy will work.

Reliability is especially important in some appli- cations, such as safety-critical systems. If we wait until we have a running system and try to eliminate the bugs, we will be too late—we won't find enough bugs, it will be too expensive to fix them, and it will take too long as well. Another set of challenges comes from the characteristics of the components and systems them- selves. If workstation programming is like assembling a machine on a bench, then embedded system design is often more like working on a car—cramped, delicate, and difficult. Let's consider some ways in which the nature of embedded computing machines makes their design more difficult.

- *Complex testing:* Exercising an embedded system is generally more difficult than typing in some data. We may have to run a real machine in order to generate the proper data. The timing of data is often important, meaning that we cannot separate the testing of an embedded computer from the machine in which it is embedded.

- *Limited observability and controllability:* Embedded computing systems usually do not come with keyboards and screens.This makes it more difficult to see what is going on and to affect the system's operation. We may be forced to watch the values of electrical signals on the microprocessor bus, for example, to know what is going on inside the system. Moreover, in real-time applica- tions we may not be able to easily stop the system to see what is going on inside.

- *Restricted development environments:* The development environments for embedded systems (the tools used to develop software and hardware) are often much more limited than those available for PCs and workstations. We generally compile code on one type of machine, such as a PC, and download it onto the embedded system. To debug the code, we must usually rely on pro- grams that run on the PC or workstation and then look inside the embedded system.

1.1.6 Performance in Embedded Computing

Embedded system designers, in contrast, have a very clear performance goal in mind—their program must meet its ***deadline***. At the heart of embedded computing is ***real-time computing***, which is the science and art of programming to deadlines. The program receives its input data; the deadline is the time at which a computation must be finished. If the program does not produce the required output by the deadline, then the program does not work, even if the output that it eventually produces is functionally correct.

- *CPU:* The CPU clearly influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.

- *Platform:* The platform includes the bus and I/O devices. The platform com- ponents that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.

- *Program:* Programs are very large and the CPU sees only a small window of the program at a time. We must consider the structure of the entire program to determine its overall behavior.

- *Task:* We generally run several programs simultaneously on a CPU, creating a ***multitasking system***. The tasks interact with each other in ways that have profound implications for performance.

- *Multiprocessor:* Many embedded systems have more than one processor— they may include multiple programmable CPUs as well as accelerators. Once again, the interaction between these processors adds yet more complexity to the analysis of overall system performance.

1.2 THE EMBEDDED SYSTEM DESIGN PROCESS

A design methodology is important for three reasons. First, it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as optimizing ***performance*** or perform- ing functional tests. Second, it allows us to develop computer-aided design tools. Developing a single program that takes in a concept for an embedded system and emits a completed design would be a daunting task, but by first breaking the process into manageable steps, we can work on automating (or at least semiautomating) the steps one at a time. Third, a design methodology makes it much easier for members of a design team to communicate. By defining the overall process, team members can more easily understand what they are supposed to do, what they should receive from other team members at certain times, and what they are to hand off when they complete their assigned steps. Since most embedded systems are designed by teams, coordination is perhaps the most important role of a well-defined design methodology.

specification, we create a more detailed description of what we want. But the specification states only how the system behaves, not how it is built. The details of the system's internals begin to take shape when we develop the architecture, which gives the system structure in terms of large components. Once we know the components we need, we can design those components, including both software modules and any specialized hardware we need. Based on those components, we can finally build a complete system.

In this section we will consider design from the ***top–down***—we will begin with the most abstract description of the system and conclude with concrete details. The alternative is a **bottom–up** view in which we start with components to build a system. Bottom–up design steps are shown in the figure as dashed-line arrows. We need bottom–up design because we do not have perfect insight into how later stages of the design process will turn out. Decisions at one stage of design are based upon estimates of what will happen later: How fast can we make a particular function run? How much memory will we need? How much system bus capacity do we need? If our estimates are inadequate, we may have to backtrack and amend our original decisions to take the new facts into account. In general, the less experience we have with the design of similar systems, the more we will have to rely on bottom-up design information to help us refine the system

But the steps in the design process are only one axis along which we can view embedded system design. We also need to consider the major goals of the design:

- manufacturing cost;
- performance (both overall speed and deadlines); and
- power consumption.

We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail:

- We must *analyze* the design at each step to determine how we can meet the specifications.
- We must then *refine* the design to add detail.

- And we must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

1.2.1 Requirements

Clearly, before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components. We generally proceed in two phases: First, we gather an informal description from the customers known as requirements, and we refine the requirements into a specification that contains enough information to begin designing the system architecture

- *Performance:* The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost. As we have noted, perfor- mance may be a combination of soft performance metrics such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed.

- *Cost:* The target cost or purchase price for the system is almost always a consideration. Cost typically has two major components: ***manufacturing cost*** includes the cost of components and assembly; ***nonrecurring engi- neering*** **(NRE)** costs include the personnel and other costs of designing the system.

- *Physical size and weight:* The physical aspects of the final system can vary greatly depending upon the application. An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict limitations on weight. A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.

- *Power consumption:* Power, of course, is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life—the customer is unlikely to be able to describe the allowable wattage.

Validating a set of requirements is ultimately a psychological task since it requires understanding both what people want and how they communicate those needs. One good way to refine at least the user interface portion of a system's requirements is to build a ***mock-up***. The mock-up may use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation. But it should give the customer a good idea of how the system will be used and how the user can react to it. Physical, nonfunctional models of devices can also give customers a better idea of characteristics such as size and weight.

shows a sample ***requirements form*** that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of the system. Let's consider the entries in the form:

- *Name:* This is simple but helpful. Giving a name to the project not only sim- plifies talking about it to other people but can also crystallize the purpose of the machine.

- *Purpose:* This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don’t understand it well enough.

- *Inputs and outputs:* These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:

— *Types of data:* Analog electronic signals? Digital data? Mechanical inputs?

— *Data characteristics:* Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?

— *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?

- *Functions:* This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?

- *Performance:* Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time frame. It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.

- *Manufacturing cost:* This includes primarily the cost of the hardware compo- nents. Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range. Cost has a substantial influence on architecture: A machine that is meant to sell at $10 most likely has a very different internal structure than a $100 system.

- *Power:* Similarly, you may have only a rough idea of how much power the system can consume, but a little information can go a long way. Typically, the most important decision is whether the machine will be battery powered or plugged into the wall. Battery-powered machines must be much more careful about how they spend energy.

- *Physical size and weight:* You should give some indication of the physical size of the system to help guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel- mounted voice recorder.

A more thorough requirements analysis for a large system might use a form similar to Figure 1.2 as a summary of the longer requirements document. After an introductory section containing this form, a longer requirements document could include details on each of the items mentioned in the introduction. For example, each individual feature described in the introduction in a single sentence may be described in detail in a section of the specification.

After writing the requirements, you should check them for internal consistency: Did you forget to assign a function to an input or output? Did you consider all the modes in which you want the system to operate? Did you place an unrealistic number of features into a battery-powered, low-cost machine?

To practice the capture of system requirements, Example 1.1 creates the requirements for a GPS moving map system.

Example:1.1 Requirements analysis of a GPS moving map

The moving map is a handheld device that displays for the user a map of the terrain around the user's current position; the map display changes as the user and the map device change posi- tion. The moving map obtains its position from the GPS, a satellite-based navigation system. The moving map display might look something like the following figure.

- *Functionality:* This system is designed for highway driving and similar uses, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.

- *User interface:* The screen should have at least 400 600 pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed to allow the user to make selections to control the system.

- *Performance:* The map should scroll smoothly. Upon power-up, a display should take no more than one second to appear, and the system should be able to verify its position and display the current map within 15 s.

■ *Cost:* The selling cost (street price) of the unit should be no more than $100.

■ *Physical size and weight:* The device should fit comfortably in the palm of the hand.

■ *Power consumption:* The device should run for at least eight hours on four AA

batteries.

1.2.2 Specification

The specification is more precise—it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it accurately reflects the customer's requirements and does so in a way that can be clearly followed during design.

Specification is probably the least familiar phase of this methodology for neo- phyte designers, but it is essential to creating working systems with a minimum of designer effort. Designers who lack a clear idea of what they want to build when they begin typically make faulty assumptions early in the process that aren't obvi- ous until they have a working system. At that point, the only solution is to take the machine apart, throw away some of it, and start again. Not only does this take a lot of extra time, the resulting system is also very likely to be inelegant, kludgey, and bug-ridden.

The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer. It should also be unambiguous enough that designers know what they need to build. Designers

1.2.3 Architecture Design

The specification does not say how the system does things, only what the system does. Describing how the system implements those functions is the purpose of the architecture. The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture. The creation of the architecture is the first phase of what many designers think of as design.

To understand what an architectural description is, let's look at a sample archi- tecture for the moving map of Example 1.1. Figure 1.3 shows a sample system architecture in the form of a ***block diagram*** that shows major operations and data flows among them.

This block diagram is still quite abstract—we have not yet specified which oper- ations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on. The diagram does, however, go a long way toward describing how to implement the functions described in the specification. We clearly see, for example, that we need to search the topographic database and to render (i.e., draw) the results for the display. We have chosen to separate those functions so that we can potentially do them in parallel—performing rendering separately from searching the database may help us update the screen more fluidly.

1.2.4 Designing Hardware and Software Components

The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and spec- ification. The components will in general include both hardware—FPGAs, boards, and so on—and software modules.

Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other com- ponents. In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component. We can also make use of standard software modules. One good example is the topographic database. Standard topographic databases exist, and you probably want to use stan- dard routines to access the database—not only is the data in a predefined format, but it is highly compressed to save storage. Using standard software for these access functions not only saves us design time, but it may give us a faster implementation for specialized functions such as the data decompression phase.

1.2.5 System Integration

System integration is difficult because it usually uncovers problems. It is often hard to observe the system in sufficient detail to determine exactly what is wrong— the debugging facilities for embedded systems are usually much more limited than what you would find on desktop systems. As a result, determining why things do not stet work correctly and how they can be fixed is a challenge in itself.

1.3 FORMALISMS FOR SYSTEM DESIGN

As mentioned in the last section, we perform a number of different design tasks at different levels of abstraction throughout this book: creating requirements and specifications,architecting the system,designing code,and designing tests. It is often helpful to conceptualize these tasks in diagrams. Luckily, there is a visual language that can be used to capture all these design tasks: the ***Unified Modeling Language (UML)*** [Boo99, Pil05]. UML was designed to be useful at many levels of abstraction in the design process.

UML is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction.

UML is an ***object-oriented*** modeling language. We will see precisely what we mean by an object in just a moment, but object-oriented design emphasizes two concepts of importance:

- It encourages the design to be described as a number of interacting objerather than a few large monolithic blocks of code

- At least some of those objects will correspond to real pieces of software or hardware in the system. We can also use UML to model the outside world that interacts with our system, in which case the objects may correspond to people or other machines. It is sometimes important to implement something we think of at a high level as a single object using several distinct pieces of code or to otherwise break up the object correspondence in the implementations.

Object-oriented (often abbreviated OO) specification can be seen in two complementary ways:

- Object-oriented specification allows a system to be described in a way that closely models real-world objects and their interactions.

- Object-oriented specification provides a basic set of primitives that can be used to describe systems with particular attributes, irrespective of the relationships of those systems' components to real-world objects.

Both views are useful. At a minimum, object-oriented specification is a set of linguistic mechanisms. In many cases, it is useful to describe a system in terms of real-world analogs. However, performance, cost, and so on may dictate that we change the specification to be different in some ways from the real-world elements we are trying to model and implement. In this case, the object-oriented specification mechanisms are still useful.

What is the relationship between an object-oriented specification and an object- oriented programming language (such as C++ [Str97])? A specification language may not be executable. But both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

Unified Modeling Language (UML)—the acronym is the name is a large lan- guage, and covering all of it is beyond the scope of this book. In this section, we introduce only a few basic concepts. In later chapters, as we need a few more UML concepts, we introduce them to the basic modeling elements introduced here. Because UML is so rich, there are many graphical elements in a UML diagram. It is important to be careful to use the correct drawing to describe something—for instance, UML distinguishes between arrows with open and filled-in arrowheads, and solid and broken lines. As you become more familiar with the language, uses of the graphical primitives will become more natural to you.

1.3.1 Structural Description

By ***structural description***, we mean the basic components of the system; we will learn how to describe how these components act in the next section. The principal component of an object-oriented design is, naturally enough, the ***object*** . An object includes a set of ***attributes*** that define its internal state. When implemented in a programming language, these attributes usually become variables or constants held in a data structure. In some cases, we will add the type of the attribute after A class is a form of type definition—all objects derived from the same class have the same characteristics, although their attributes may have different values. A class defines the attributes that an object may have. It also defines the ***operations*** that determine how the object interacts with the rest of the world. In a programming language, the operations would become pieces of code used to manipulate the object. The UML description of the *Display* class is shown in Figure 1.6. The class has the name that we saw used in the *d* 1 object since *d* 1 is an instance of class *Display*. The *Display* class defines the *pixels* attribute seen in the object; remember that when we instantiate the class an object, that object will have its own memory so that different objects of the same class have their own values for the attributes. Other classes can examine and modify class attributes; if we have to do something more complex than use the attribute directly, we define a behavior to perform that function.

A class defines both the ***interface*** for a particular type of object and that object's ***implementation***. When we use an object, we do not directly manipulate its attributes—we can only read or modify the object's state through the operations that define the interface to the object. (The implementation includes both the attributes and whatever code is used to implement the operations.) As long as we do not change the behavior of the object seen at the interface, we can change the implementation as much as we want. This lets us improve the system by, for example, speeding up an operation or reducing the amount of memory required without requiring changes to anything else that uses the object.

There are several types of ***relationships*** that can exist between objects and classes:

- ***Association*** occurs between objects that communicate with each other but have no ownership relationship between them.

- ***Aggregation*** describes a complex object made of smaller objects.

- ***Composition*** is a type of aggregation in which the owner does not allow access to the component objects.

- ***Generalization*** allows us to define one class in terms of another.

Unified Modeling Language, like most object-oriented languages, allows us to define one class in terms of another. An example is shown in Figure 1.7, where we ***derive*** two particular types of displays. The first, *BW_display,* describes a black- and-white display. This does not require us to add new attributes or operations, but we can specialize both to work on one-bit pixels. The second, *Color_map_display*, uses a graphic device known as a color map to allow the user to select from a behaviors—for example, large number of available colors even with a small number of bits per pixel. This class defines a *color_map* attribute that determines how pixel values are mapped onto display colors. A ***derived class*** inherits all the attributes and operations from its ***base class***. In this class, *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class. This relation is transitive—if *Display* were derived from another class, both *BW_display* and *Color_map_display* would inherit all the attributes and operations of *Display's* base class as well. Inheritance has two purposes. It of course allows us to succinctly describe one class that shares some characteristics with another class. Even more important, it captures those relationships between classes and documents them. If we ever need to change any of the classes, knowledge of the class structure helps us determine the reach of changes—for example, should the change affect only *Color_map_display* objects or should it change all Display objects?

Unified Modeling Language considers inheritance to be one form of general- ization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead. Both *BW_display* and *Color* versions of *Display*, so *Display* generalizes both of them. UML also allows us to define ***multiple inheritance***, in which a class is derived from more than one base class. (Most object-oriented programming languages support multiple inheritance as well.) An example of multiple inheritance is shown in Figure 1.8; we have omit- ted the details of the classes' attributes and operations for simplicity. In this case, we have created a *Multimedia_display* class by combining the *Display* class with a *Speaker* class for sound. The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*. Because multiple inheritance causes the sizes of the attribute set and operations to expand so quickly, it should be used with care.

A ***link*** describes a relationship between objects; association is to link as class is to object. We need links because objects often do not stand alone; associations let us capture type information about these links. examples of links and an association. When we consider the actual objects in the system, there is a set of messages that keeps track of the current number of active messages (two in this example) and points to the active messages. In this case, the link defines the *contains* relation. When generalized into classes, we define an association between the message set class and the message class. The association is drawn as a line between the two labeled with the name of the association, namely, *contains*. The ball and the number at the message class end indicate that the message message objects. Sometimes we may want to attach data to the links themselves; we can specify this in the association by attaching a class-like box to the association's edge, which holds the association's data.

Typically, we find that we use a certain combination of elements in an object or class many times. We can give these patterns names, which are called ***stereotypes***

1.3.2 Behavioral Description

We have to specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a

state machine. Figure 1.10 shows UML states; the transition between two states is shown by a skeleton arrow.

These state machines will not rely on the operation of a clock, as in hardware;

rather, changes from one state to another are triggered by the occurrence of ***events***.

■ A ***signal*** is an asynchronous occurrence. It is defined in UML by an object that is labeled as a <<*signal*>>. The object in the diagram serves as a declaration of the event's existence. Because it is an object, a signal may have parameters that are passed to the signal's receiver.

■ A ***call event*** follows the model of a procedure call in a programming language.

■ A ***time-out event*** causes the machine to leave a state after a certain amount of time. The label *tm(time-value)* on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism.

We show the occurrence of all types of signals in a UML diagram in the same way—

Let's consider a simple state machine specification to understand the semantics of UML state machines. A state machine for an operation of the display is shown in Figure 1.12. The start and stop states are special states that help us to organize the flow of the state machine. The states in the state machine represent different conceptual operations. In some cases, we take conditional transitions out of states based on inputs or the results of some computation done in the state. In other cases, we make an unconditional transition to the next state. Both the unconditional and conditional transitions make use of the call event. Splitting a complex operation into several states helps document the required steps, much as subroutines can be used to structure code.

It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved. In this case, we can create a sequence diagram, like the one for a mouse click scenario shown in Figure 1.13. A ***sequence diagram*** is somewhat similar to a hardware timing diagram, although the time flows verti- cally in a sequence diagram, whereas time typically flows horizontally in a timing diagram. The sequence diagram is designed to show a particular scenario or choice of events—it is not convenient for showing a number of mutually exclusive possibil- ities. In this case, the sequence shows what happens when a mouse click is on the menu region. Processing includes three objects shown at the top of the diagram. Extending below each object is its *lifeline*, a dashed line that shows how long the object is alive. In this case, all the objects remain alive for the entire sequence, but in other cases objects may be created or destroyed during processing. The boxes along the lifelines show the *focus of control* in the sequence,

that is, when the object is actively processing. In this case, the mouse object is active only long enough to create the *mouse_click* event. The display object remains in play longer; it in turn uses call events to invoke the menu object twice: once to determine which menu item was selected and again to actually execute the menu call. The find_region( ) call is internal to the display object, so it does not appear as an event in the diagram.

1.4 MODEL TRAIN CONTROLLER

In order to learn how to use UML to model systems, we will specify a simple system, a model train controller, which is illustrated in Figure 1.14. The user sends messages to the train with a control box attached to the tracks. The control box may have familiar controls such as a throttle, emergency stop button, and so on. Since the train receives its electrical power from the two rails of the track, the control box can send

UNIT -2

Instruction Set CPUs

Harvard architectures are widely used today for one very simple reason—the separation of program and data memories provides higher performance for digital signal processing. Processing signals in real-time places great strains on the data access system in two ways: First, large amounts of data flow through the CPU; and second, that data must be processed at precise intervals, not just when the CPU gets around to it. Data sets that arrive continuously and periodically are called ***streaming data***. Having two memories with separate ports provides higher memory band- width; not making data and memory compete for the same port also makes it easier to move the data at the proper times. DSPs constitute a large fraction of all micro- processors sold today, and most of them are Harvard architectures. A single example shows the importance of DSP: Most of the telephone calls in the world go through at least two DSPs, one at each end of the phone call.

Another axis along which we can organize computer architectures relates to their instructions and how they are executed. Many early computer architectures were what is known today as ***complex instruction set computers (CISC)***. These machines provided a variety of instructions that may perform very com- plex tasks, such as string searching; they also generally used a number of different instruction formats of varying lengths. One of the advances in the development of high-performance microprocessors was the concept of ***reduced instruction set computers (RISC)***. These computers tended to provide somewhat fewer and sim- pler instructions. The instructions were also chosen so that they could be efficiently executed in ***pipelined*** processors. Early RISC designs substantially outperformed CISC designs of the period. As it turns out, we can use RISC techniques to efficiently execute at least a common subset of CISC instruction sets, so the performance gap between RISC-like and CISC-like instruction sets has narrowed somewhat.

Beyond the basic RISC/CISC characterization, we can classify computers by sev- eral characteristics of their instruction sets. The instruction set of the computer defines the interface between software modules and the underlying hardware; the instructions define what the hardware will do under certain circumstances. Instructions can have a variety of characteristics, including:

∎ Fixed versus variable length.

∎ Addressing modes.

∎ Numbers of operands.

∎ Types of operations supported.

The set of registers available for use by programs is called the ***programming model*** ,also known as the ***programmer model*** . (The CPU has many other registers that are used for internal operations and are unavailable to programmers.)

There may be several different implementations of an architecture. In fact, the architecture definition serves to define those characteristics that must be true of all implementations and what may vary from implementation to implementation. Different CPUs may offer different clock speeds, different cache configurations, changes to the bus or interrupt lines, and many other changes that can make one model of CPU more attractive than another for any given application.

2.1.2 Assembly Language

Figure 2.3 shows a fragment of ARM assembly code to remind us of the basic features of assembly languages. Assembly languages usually share the same basic features:

■ One instruction appears per line.

■ ***Labels***, which give names to memory locations, start in the first column.

■ Instructions must start in the second column or after to distinguish them from labels.

■ Comments run from some designated comment character (; in the case of

ARM) to the end of the line.

Assembly language follows this relatively structured form to make it easy for the ***assembler*** to parse the program and to consider most aspects of the program line by line. (It should be remembered that early assemblers were writ- ten in assembly language to fit in a very small amount of memory. Those early restrictions have carried into modern assembly languages by tradition.) Figure 2.4 shows the format of an ARM data processing instruction such as an ADD. For the instruction

```
ADDGT r0,r3,#5
```

the *cond* field would be set according to the GT condition (1100), the *opcode* field would be set to the binary code for the ADD instruction (0100), the first *operand* register *Rn* would be set to 3 to represent r3, the destination register *Rd* would be set to 0 for r0, and the *operand 2* field would be set to the immediate value of 5.

Assemblers must also provide some ***pseudo-ops*** to help programmers create complete assembly language programs. An example of a pseudo-op is one that allows data values to be loaded into memory locations. These allow constants, for example, to be set into memory. An example of a memory allocation pseudo-op for ARM is shown in Figure 2.5. The ARM % pseudo-op allocates a block of memory of the size specified by the operand and initializes those locations to zero.

```
                label1    ADR r4,c
LDR r0,[r4]          ; a comment
                          ADR r4,d
                          LDR r1,[r4]
SUB r0,r0,r1         ; another comment
```

FIGURE 2.3

An example of ARM assembly language

2.2.1 Processor and Memory Organization

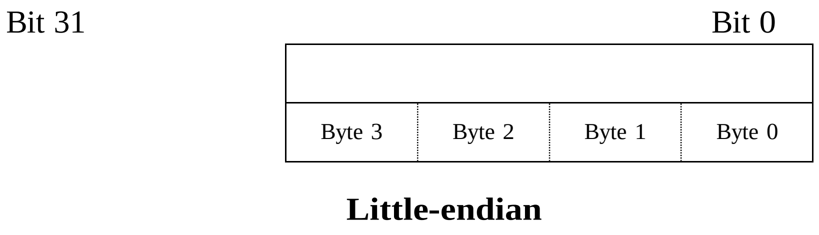
Different versions of theARM architecture are identified by different numbers. ARM7 is a von Neumann architecture machine, while ARM9 uses a Harvard architecture. However, this difference is invisible to the assembly language programmer, except for possible performance differences.

The ARM architecture supports two basic types of data:

■ The standard ARM word is 32 bits long.

■ The word may be divided into four 8-bit bytes.

ARM7 allows addresses up to 32 bits long. An address refers to a byte, not a word. Therefore, the word 0 in the ARM address space is at location 0, the word 1 is at 4, the word 2 is at 8, and so on. (As a result, the PC is incremented by 4 in the absence of a branch.) The ARM processor can be configured at power-up to address the bytes in a word in either ***little-endian*** mode (with the lowest-order byte residing in the low-order bits of the word) or ***big-endian*** mode (the lowest-order byte stored in the highest bits of the word), as illustrated in Figure 2.6 [Coh81].



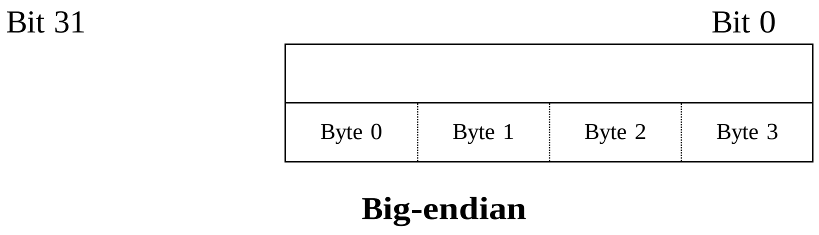


FIGURE 2.6

Byte organizations within an ARM word.

2.2.2 Data Operations

Arithmetic and logical operations in C are performed in variables. Variables are implemented as memory locations. Therefore, to be able to write instructions to perform C expressions and assignments, we must consider both arithmetic and logical instructions as well as instructions for reading and writing memory.

Figure 2.7 shows a sample fragment of C code with data declarations and several assignment statements. The variables *a, b, c, x, y,* and *z* all become data locations in memory. In most cases data are kept relatively separate from instructions in the program's memory image.

In the ARM processor, arithmetic and logical operations cannot be performed directly on memory locations. While some processors allow such operations to directly reference main memory, ARM is a ***load-store architecture***—data operands must first be loaded into the CPU and then stored back to main memory to save the results. Figure 2.8 shows the registers in the basic ARM programming model. ARM has 16 general-purpose registers, r0 through r15. Except for r15, they are identical—any operation that can be done on one of them can be done on the other one also. The r15 register has the same capabilities as the other registers, but it is also used as the program counter. The program counter should of course not be overwritten for use in data operations. However, giving the PC the properties of a general-purpose register allows the program counter value to be used as an operand in computations, which can make certain programming tasks easier.

The other important basic register in the programming model is the ***cur- rent program status register (CPSR)***. This register is set automatically during every arithmetic, logical, or shifting operation. The top four bits of the CPSR hold the following useful information about the results of that arithmetic/logical operation:

- ■ The negative (N) bit is set when the result is negative in two's-complement arithmetic.

- ■ The zero (Z) bit is set when every bit of the result is zero.

- ■ The carry (C) bit is set when there is a carry out of the operation.

- ■ The overflow (V) bit is set when an arithmetic operation results in an overflow.

```
int a, b, c, x, y, z;

 x    (a    b)    c;

  y    a*(b    c);

z    (a << 2) | (b & 15);
```

FIGURE 2.7

A C fragment with data operations.

These bits can be used to check easily the results of an arithmetic operation. However, if a chain of arithmetic or logical operations is performed and the inter- mediate states of the CPSR bits are important, then they must be checked at each step since the next operation changes the CPSR values. Example 2.1 illustrates the computation of CPSR bits.

Example 2.1

Status bit computation in the ARM

An ARM word is 32 bits. In C notation, a hexadecimal number starts with 0x, such as 0xffffffff, which is a two's-complement representation of 1 in a 32-bit word.

Here are some sample calculations:

■ *1 1 0:* Written in 32-bit format, this becomes 0xffffffff 0x1 0x0, giving the

CPSR value of NZCV 1001.

■ *0 1 1:* 0x0 0x1 0xffffffff, with NZCV 1000.

■ 2^{31} *1 1* 2^{31}: 0x7fffffff 0x1 0x80000000, with NZCV 1001.

The basic form of a data instruction is simple:

```
ADD r0,r1,r2
```

This instruction sets register r0 to the sum of the values stored in r1 and r2. In addition to specifying registers as sources for operands, instructions may also provide ***immediate operands***, which encode a constant value directly in the instruction. For example,

```
ADD r0,r1,#2
```

sets r0 to r1 2.

The major data operations are summarized in Figure 2.9. The arithmetic opera- tions perform addition and subtraction; the with-carry versions include the current value of the carry bit in the computation. RSB performs a subtraction with the order of the two operands reversed, so that RSB r0, r1, r2 sets r0 to be r2 r1. The bit-wise logical operations perform logical AND, OR, and XOR operations (the exclusive or is called EOR). The BIC instruction stands for bit clear: BIC r0, r1, r2 sets r0 to r1 and not r2. This instruction uses the second source operand as a mask: Where a bit in the mask is 1, the corresponding bit in the first source operand is cleared. The MUL instruction multiplies two values, but with some restrictions: No operand may be an immediate, and the two source operands must be different registers. The MLA instruction performs a multiply-accumulate operation, particularly useful in matrix operations and signal processing. The instruction

MLA r0,r1,r2,r3

sets r0 to the value r1 r2 r3.

The shift operations are not separate instructions—rather, shifts can be applied to arithmetic and logical instructions. The shift modifier is always applied to the second source operand. A left shift moves bits up toward the most-significant bits, while a right shift moves bits down to the least-significant bit in the word. The LSL and LSR modifiers perform left and right logical shifts, filling the least-significant bits of the operand with zeroes. The arithmetic shift left is equivalent to an LSL, but the ASR copies the sign bit—if the sign is 0, a 0 is copied, while if the sign is 1, a

1 is copied. The rotate modifiers always rotate right, moving the bits that fall off the least-significant bit up to the most-significant bit in the word. The RRX modifier performs a 33-bit rotate, with the CPSR's C bit being inserted above the sign bit of the word; this allows the carry bit to be included in the rotation.

stored in the register is used as the address to be fetched from memory; the result of that fetch is the desired operand value. Thus, as illustrated in Figure 2.13, if we set r1 0 100, the instruction

LDR r0,[r1]

sets r0 to the value of memory location 0x100. Similarly, STR r0,[r1] would store the contents of r0 in the memory location whose address is given in r1. There are several possible variations:

LDR r0,[r1, – r2]

this step. Thus, as shown in Figure 2.14, if we give location 0x100 the name FOO, we can use the pseudo-operation

ADR r1,FOO

to perform the same function of loading r1 with the address 0x100.

Example 2.2 illustrates how to implement C assignments in ARM instruction.

Example 2.2

C assignments in ARM instructions

We will use the assignments of Figure 2.7. The semicolon (;) begins a comment after an instruction, which continues to the end of that line. The statement

$$x \quad (a \quad b) \quad c;$$

can be implemented by using r0 for *a*, r1 for *b*, r2 for *c*, and r3 for *x*. We also need registers for indirect addressing. In this case, we will reuse the same indirect addressing register, r4, for each variable load. The code must load the values of *a*, *b*, and *c* into these registers before performing the arithmetic, and it must store the value of *x* back to memory when it is done. This code performs the following necessary steps:

```
ADR r4,a      ; get address for a
LDR r0,[r4]   ; get value of a
ADR r4,b      ; get address for b, reusing r4
LDR r1,[r4]   ; load value of b
ADD r3,r0,r1  ; set intermediate result for x to a + b
ADR r4,c      ; get address for c
LDR r2,[r4]   ; get value of c
SUB r3,r3,r2  ; complete computation of x
ADR r4,x      ; get address for x

STR r3,[r4]   ; store x at proper location
```

```
LDR r0,[r4] ; get value of a
MUL r2,r2,r0 ; compute final value of y
ADR r4,y     ; get address for y
STR r2,[r4] ; store value of y at proper location
```

2.2.3 Flow of Control

The B (branch) instruction is the basic mechanism in ARM for changing the flow of control. The address that is the destination of the branch is often called the ***branch target***. Branches are ***PC-relative***—the branch specifies the offset from the current PC value to the branch target. The offset is in words, but because the ARM is byte- addressable, the offset is multiplied by four (shifted left two bits, actually) to form a byte address. Thus, the instruction

```
B #100
```

will add 400 to the current PC value.

We often wish to branch conditionally,based on the result of a given computation. The if statement is a common example. The ARM allows *any* instruction, including branches, to be executed conditionally. This allows branches to be conditional, as well as data operations. Figure 2.15 summarizes the condition codes.

Example 2.3

Implementing an* if *statement in ARM

We will use the following if statement as an example:

```
if (a < b) {
x = 5;
y = c + d;
}
else x = c - d;
```

The implementation uses two blocks of code, one for the true case and another for the false case. A branch may either fall through to the true case or branch to the false case:

```
; compute and test the condition
ADR r4,a        ; get address for a
LDR r0,[r4]     ; get value of a
ADR r4,b        ; get address for b
LDR r1,[r4]     ; get value of b
CMP r0, r1      ; compare a < b
BGE fblock      ; if a >= b, take branch
; the true block follows
```

```
        MOV r0,#5          ; generate value for x

        ADR r4,x           ; get address for x STR
rO,[r4]   ; store value of x ADR r4,c        ; get
address for c LDR rO,[r4]    ; get value of c

        ADR r4,d           ; get address for d

        LDR r1,[r4]        ; get value of d

        ADD rO,rO,r1       ; compute c + d

        ADR r4,y           ; get address for y

        STR rO,[r4]        ; store value of y

        B after            ; branch around the false
block

        ; the false block follows

        fblock ADR r4,c         ; get address for c

        LDR rO,[r4]        ; get value of c

        ADR r4,d           ; get address for d

        LDR r1,[r4]        ; get value of d

        SUB rO,rO,r1       ; compute c – d

        ADR r4,x           ; get address for x

        STR rO,[r4]        ; store value of x after
... ; code after the if statement
```

Example 2.4

Implementing the C switch statement in ARM

The switch statement in C takes the following form:

```
switch (test) { case 0: ... break; case 1:
... break;

...

}
```

The above statement could be coded like an if statement by first testing *test* *A*, then *test* *B*, and so forth. However, it can be more efficiently implemented by using base-plus-offset addressing and building what is known as a ***branch table***:

```
ADR r2,test ; get address for test

LDR r0,[r2] ; load value for test

ADR r1,switchtab ; load address for
switch table

LDR r15,[r1,r0,LSL #2]

switchtab DCD case0
```

```
DCD case1

...

case0      ... ; code for  case  0

...

case1      ... ; code for  case  1

...
```

This implementation uses the value of test as an offset into a table, where the table holds the addresses for the blocks of code that implement the various cases. The heart of this code is the LDR instruction, which packs a lot of functionality into a single instruction:

■ It shifts the value of r0 left two bits to turn the offset into a word address.

■ It uses base-plus-offset addressing to add the left-shifted value of test (held in r0) to the address of the base of the table held in r1.

■ It sets the PC (r15) to the new address computed by the instruction.The loop is a very common C statement, particularly in signal processing code. Loops can be naturally implemented using conditional branches. Because addressing mode. A simple but common use of a loop is in the FIR filter, which is explained in Application Example 2.1; the loop-based implementation of the FIR filter is described in Example 2.5.