Sub Code: 10CS72 **IA Marks :25**
Hrs/Week: 04 **Exam Hours :03**
Total Hrs: 52 **Exam Marks :100**

PART- A

UNIT – 1 **6 Hours**
Embedded Computing: Introduction, Complex Systems and Microprocessors, Embedded Systems Design Process, Formalism for System design Design Example: Model Train Controller.

UNIT – 2 **7 Hours**
Instruction Sets, CPUs: Preliminaries, ARM Processor, Programming Input and Output, Supervisor mode, Exceptions, Traps, Coprocessors, Memory Systems Mechanisms, CPU Performance, CPU Power Consumption. Design Example: Data Compressor.

UNIT – 3 **6 Hours**
Bus-Based Computer Systems: CPU Bus, Memory Devices, I/O devices, Component Interfacing, Designing with Microprocessor, Development and Debugging, System-Level Performance Analysis Design Example: Alarm Clock.

UNIT – 4 **7 Hours**
Program Design and Analysis: Components for embedded programs, Models of programs, Assembly, Linking and Loading, Basic Compilation Techniques, Program optimization, Program-Level performance analysis, Software performance optimization, Program-Level energy and power analysis, Analysis and optimization of program size, Program validation and testing. Design Example: Software modem.

PART- B

UNIT – 5 **6 Hours**
Real Time Operating System (RTOS) Based Design – 1: Basics of OS, Kernel, types of OSs, tasks, processes, Threads, Multitasking and Multiprocessing, Context switching, Scheduling Policies, Task Communication, Task Synchronization.

UNIT – 6 **6 Hours**
RTOS-Based Design - 2: Inter process Communication mechanisms, Evaluating OS performance, Choice of RTOS, Power Optimization. Design Example: Telephone Answering machine.

UNIT – 7 **7 Hours**
Distributed Embedded Systems: Distributed Network Architectures, Networks for Embedded Systems: I2C Bus, CAN Bus, SHARC Link Ports, Ethernet, Myrinet, Internet, Network Based Design. Design Example: Elevator Controller.

UNIT – 8 **7 Hours**
Embedded Systems Development Environment: The Integrated Development Environment, Types of File generated on Cross Compilation, Dis-assembler /Decompiler, Simulators, Emulators, and Debugging, Target Hardware Debugging.

Text Books:

1. Wayne Wolf: Computers as Components, Principles of Embedded Computing Systems Design, 2nd Edition, Elsevier, 2008.
2. Shibu K V: Introduction to Embedded Systems, Tata McGraw Hill, 2009 (Chapters 10, 13)

Reference Books:

1. James K. Peckol: Embedded Systems, A contemporary Design Tool, Wiley India, 2008
2. Tammy Neorgaard: Embedded Systems Architecture, Elsevier, 2005.

Contents

UNIT 1

Embedded Computing

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.

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.

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.

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.

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 properties 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.

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 dissipation. 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 applications, 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 applications 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.

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 components 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.

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.

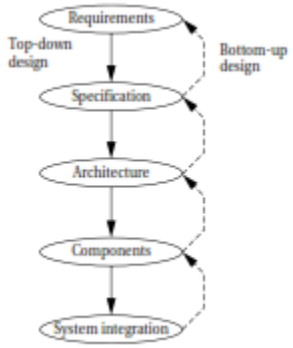


Fig.1.1Major levels of abstraction in the design process.

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.

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 simplifies 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 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 simplifies 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

■ *Manufacturing cost:* This includes primarily the cost of the hardware components. 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.

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 obvious 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

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. Figure 1.2 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 operations 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.

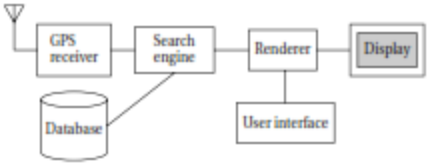


Fig. 1.2 Block diagram for the moving map.

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 specification. 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 components. 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 standard 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.

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.

FORMALISMS FOR SYSTEM DESIGN

As mentioned in the last section, we perform a number of different design tasks at different levels of abstraction :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)*** 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 object 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.

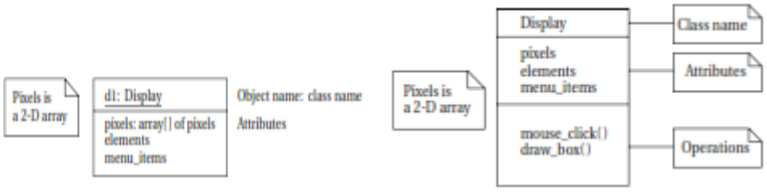


Fig.1.4An object in UML notation. A class in UML notation.

■ 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 language, 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.

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 generalization. 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 inheritances***, 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***

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 vertically 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 possibilities. 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.

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 1.5 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 signals to the train over the tracks by modulating the power supply voltage. As shown in the figure, the control panel sends packets over the tracks to the receiver on the train. The train includes analog electronics to sense the bits being transmitted and a control system to set the train motor's speed and direction based on those commands. Each packet includes an address so that the console can control several trains on the same track; the packet also includes an error correction code (ECC) to guard against transmission errors. This is a one-way communication system—the model train cannot send commands back to the user.

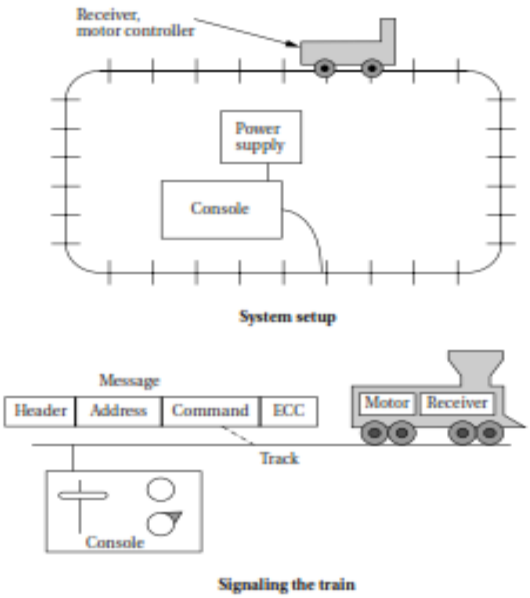


Fig.1.5A model train control system.

Requirements

Before we can create a system specification, we have to understand the require- ments. Here is a basic set of requirements for the system:

■ The console shall be able to control up to eight trains on a single track.

■ The speed of each train shall be controllable by a throttle to at least 63 different levels in each direction (forward and reverse).

There shall be an inertia control that shall allow the user to adjust the respon- siveness of the train to commanded changes in speed. Higher inertia means that the train responds more slowly to a change in the throttle, simulating the inertia of a large train. The inertia control will provide at least eight different levels.

■ There shall be an emergency stop button.

■ An error detection scheme will be used to transmit messages.

Name Model train controller Purpose Control speed of up to eight model trains Inputs Throttle, inertia setting, emergency stop, train number Outputs Train control signals Set engine speed based upon inertia settings; respond to emergency stop Performance Can update train speed at least 10 times per second Manufacturing cost $50 Power 10 W (plugs into wall) size and weight Console should be comfortable for two hands, approximate size of standard keyboard; weight 2 pounds We will develop our system using a widely used standard for model train control. We could develop our own train control system from scratch, but basing our system upon a standard has several advantages in this case: It reduces the amount of work we have to do and it allows us to use a wide variety of existing trains and other pieces of equipment. **1.4.2 DCC** The **Digital Command Control (DCC)** standard (http://www.nmra.org/ standards/DCC/standards_rps/DCCStds.html) was created by the National Model Railroad Association to support interoperable digitally-controlled model trains. Hob- byists started building homebrew digital control systems in the 1970s and Marklin developed its own digital control system in the 1980s. DCC was created to provide a standard that could be

built by any manufacturer so that hobbyists could mix and match components from multiple vendors. The DCC standard is given in two documents: Standard S-9.1, the DCC Electrical Standard, defines how bits are encoded on the rails for transmission.

■ Standard S-9.2, the DCC Communication Standard, defines the packets that carry information Any DCC-conforming device must meet these specifications. DCC also provides several recommended practices. These are not strictly required but they provide some hints to manufacturers and users as to how to best use DCC. The DCC standard does not specify many aspects of a DCC train system. It doesn't define the control panel, the type of microprocessor used, the programming lan- guage to be used, or many other aspects of a real model train system. The standard concentrates on those aspects of system design that are necessary for interoper- ability. Overstandardization, or specifying elements that do not really need to be standardized, only makes the standard less attractive and harder to implement. The Electrical Standard deals with voltages and currents on the track. While the electrical engineering aspects of this part of the specification are beyond the scope of the book, we will briefly discuss the data encoding here. The standard must be carefully designed because the main function of the track is to carry power to the locomotives. The signal encoding system should not interfere with power transmission either to DCC or non-DCC locomotives. A key requirement is that the data signal should not change the DC value of the rails. The data signal swings between two voltages around the power supply volt- age. As shown in Figure 1.15, bits are encoded in the time between transitions, not by voltage levels. A 0 is at least 100 s while a 1 is nominally 58 s. The

dura- tions of the high (above nominal voltage) and low (below nominal voltage) parts of a bit are equal to keep the DC value constant. The specification also gives the allowable variations in bit times that a conforming DCC receiver must be able to tolerate. The standard also describes other electrical properties of the system, such as allowable transition times for signals. The DCC Communication Standard describes how bits are combined into packets and the meaning of some important packets. Some packet types are left undefined in the standard but typical uses are given in Recommended Practices documents.

■ *P* is the preamble, which is a sequence of at least 10 1 bits. The command station should send at least 14 of these 1 bits, some of which may be corrupted during transmission.

■ *S* is the packet start bit. It is a 0 bit.

■ *A* is an address data byte that gives the address of the unit, with the most significant bit of the address transmitted first. An address is eight bits long. The addresses 00000000, 11111110, and 11111111 are reserved.
■ *s* is the data byte start bit, which, like the packet start bit, is a 0.
■ *D* is the data byte, which includes eight bits. A data byte may contain an address, instruction, data, or error correction information.
■ *E* is a packet end bit, which is a 1 bit. A packet includes one or more data byte start bit/data byte combinations. Note that the address data byte is a specific type
of data byte. A ***baseline packet*** is the minimum packet that must be accepted by all DCC implementations. More complex packets are given in a Recommended Practice doc- ument. A baseline packet has three data bytes: an address data byte that gives the intended receiver of the packet; the instruction data byte provides a basic instruc- tion; and an error correction data byte is used to detect and correct transmission errors. The instruction data byte carries several pieces of information. Bits 0–3 provide a 4-bit speed value. Bit 4 has an additional speed bit, which is interpreted as the least significant speed bit. Bit 5 gives direction, with 1 for forward and 0 for reverse. Bits 7–8 are set at 01 to indicate that this instruction provides speed and direction. The error correction databyte is the bitwise exclusive OR of the address and instruction data bytes. The standard says that the command unit should send packets frequently since a packet may be corrupted. Packets should be separated by at least 5 ms.

UNIT 2

INSTRUCTION SETS

The instruction set is the key to analyzing the performances of programs to implement a particular function we must understand the types of instructions that your CPU provides.

HERE WE USE TWO CPU'S AS AN EXAMPLE

1. The ARM processor: is widely used in Cell phones and many other systems. (We will mainly concentrate on ARM version 7.

2. Digital signal processor (DSP): The Texas instruments C5XX.

We start with a brief introduction to the terminology of computer architecture and instruction sets

PRELIMINARIES

We will study some general concepts in computer architectures. Including the different styles of computer architecture and the nature of the assembly language (Example: Microprocessor*) **COMPUTER ARCHITECURE TAXONOMY.**

Before dealing of microprocessor instruction sets details, we review the taxonomy of the basic terminology though we can organize a computer.

A block diagram of one type of computer as shown in fig 2.1 is a **VON NEUMAN ARCHITECTURE**.

This computing system consists of central processing unit (CPU) and a memory. The memory holds both data and instructions and can be read and written when given an address. A memory that holds both data and instructions is known as VON NEUMAN MACHINE.

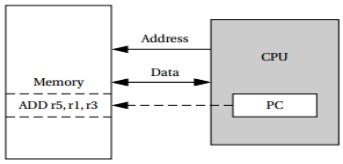


FIGURE 2.1
A von Neumann architecture computer.

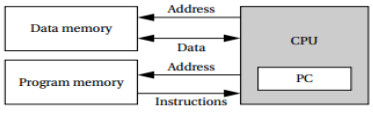


FIGURE 2.2
A Harvard architecture.

The CPU has several internal registers that stores values and uses internally one of these registers in the program called the program counter (PC). This holds the address in the memory of an instruction. The CPU fetches the instruction from the memory and decodes the instruction and then executes it.

HARVARD ARCHITECTURE

One of the alternative to the VON NEUMAN style of organizing is the computer architecture. Which is nearly as old as the VON NEUMAN architecture.

- Harvard machine has separate memories for data and program.
- The PC points to the program memory, not data memory.
- It is harder to write self-modifying programs on the Harvard machines.
- The Harvard architecture is the separation of programs and data memories provides higher performance for digital processing.

Next another two types of architecture which related to the instruction sets are as follows. The **instruction set architecture (ISA).**

1> **CISC**

2> **RISC**

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 complex 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).

One of the advancement in the development of high performance microprocessor is RISC. These instructions were chosen to execute in pipelined processors so that there could be efficient execution and the speed of execution would be faster because of pipelining. These computers tended to provide fewer and simpler 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 several 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.

INSTRUCTIONS

The purpose of an instruction is to direct the hardware of the microprocessor to perform a series of actions.

Such actions can include the perform an arithmetic or logical calculation, assign on read the value of a variable or move data from one place to another, such as input to memory from memory to output, such actions are called operations.

The entities that instructions operate on are denoted operands. Assembly language for the machine language is a collection of 0's and 1's that control the hardware components in the execution of the instructions. At assembly level we work with set of instructions that the machine supports that is the instruction sets.

ASSEMBLY LANGUAGE:

Assembly languages usually share the same basic features:

■ Only 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.) For the instruction ADDGT r0, r3, #5

the condition field would be set according to the GT condition (1100), 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. 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        → for getting the value for c
          LDR r0, [r4]     →for loading the value of c
          ADD r4, d        →get address for d, reusing r4
          LDR r1, [r4]     →load value of d
          SUB r0 , r0, r1  →set intermediate value for c and d
```

ARM (ADVANCED RISC MACHINE) PROCESSOR:

ARM instructions are written one per line starting after the first column, comments begin with a semicolon and continue to the end of the line. A label which gives a name to a memory location comes at the beginning of the line starting with the first column.

```
EXAMPLE:  LDR r0, [r8]   ; comment
          Label  ADDR r4, r0, r1
```

PROCESSOR AND MEMORY ORGANIZATION:

Different versions of the ARM 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.

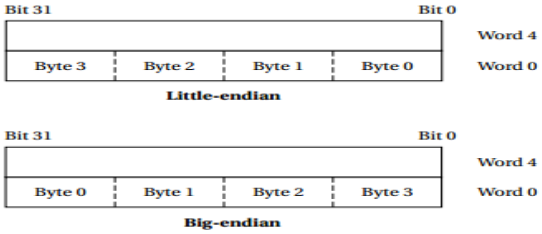
■The word may be divided into four 8-bit bytes. ARM7 allows addresses up to 32.

The ARM processor can configured into two types of modes.

→ Little Indian mode: the lowest order byte residing in the lower-order bits of word.

→Big Indian mode: the lowest order byte residing in the highest bits of word.

DIAGRAM:



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. 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.

Inta, b, c, x, y, z;

x = (a+b)-c;

y = a*(b+c);

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

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 current 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.

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 arithmetic operations 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: Nooperand 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, a1 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 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:

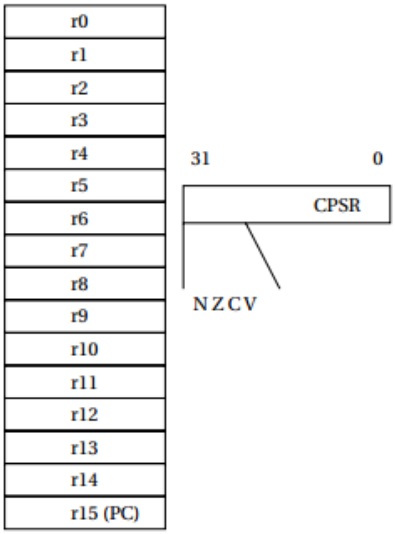
LDR r0, [r1, – r2]

Thus 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.

How to implement C assignments in ARM instruction



THE MAJOR DATA OPERATIONS IN ARM:

ADD	Add
ADC	Add with carry
SUB	Subtract
SBC	Subtract with carry
RSB	Reverse subtract
RSC	Reverse subtract with carry
MUL	Multiply
MLA	Multiply and accumulate

Arithmetic

AND	Bit-wise and
ORR	Bit-wise or
EOR	Bit-wise exclusive-or
BIC	Bit clear

Logical

LSL	Logical shift left (zero fill)
LSR	Logical shift right (zero fill)
ASL	Arithmetic shift left
ASR	Arithmetic shift right
ROR	Rotate right
RRX	Rotate right extended with C

Shift/rotate

CMP	Compare
CMN	Negated compare
TST	Bit-wise test
TEQ	Bit-wise negated test

FIGURE 2.10
ARM comparison instructions.

MOV	Move
MVN	Move negated

FIGURE 2.11
ARM move instructions.

LDR	Load
STR	Store
LDRH	Load half-word
STRH	Store half-word
LDRSH	Load half-word signed
LDRB	Load byte
STRB	Store byte
ADR	Set register to address

FIGURE 2.12
ARM load-store instructions and pseudo-operations.

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.

EQ	Equals zero	Z = 1
NE	Not equal to zero	Z = 0
CS	Carry set	C = 1
CC	Carry clear	C = 0
MI	Minus	N = 1
PL	Nonnegative (plus)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 1 and Z = 0
LS	Unsigned lower or same	C = 0 or Z = 1
GE	Signed greater than or equal	N = V
LT	Signed less than	N V
GT	Signed greater than	Z = 0 and N = V
LE	Signed less than or equal	Z = 1 or N V

FEATURES OF THE ARM INSTRUCTION SET:

- ➔ All instructions are 32 bit long
- ➔ Most instructions execute in a single cycle
- ➔ Every instructions can be conditionally executed.

A LOAD/ STORE ARCHITECTURE:

Possible to load/store multiple registers at once. Data processing instructions act only on registers.

- Three operand format
  - ADD r0, r1, r2
- Combined ALU and shift operation in a single instruction for high speed bit manipulation.
- Specific memory access instructions with powerful auto indexing addressing modes.
- 32 bit and 8 bit data types and also 16 bit data types on ARM architecture
- Flexible multiple register loads store instructions.

WRITE AN ARM ASSEMBLY CODE TO IMPLEMENT THE FOLLOWING CONDITIONAL IF:

```
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 r0,[r4]     ; store value of x
       ADR r4,c        ; get address for c
       LDR r0,[r4]     ; get value of c
       ADR r4,d        ; get address for d
       LDR r1,[r4]     ; get value of d
       ADD r0,r0,r1    ; compute c + d
       ADR r4,y        ; get address for y
       STR r0,[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 r0,[r4]     ; get value of c
       ADR r4,d        ; get address for d
       LDR r1,[r4]     ; get value of d
       SUB r0,r0,r1    ; compute c - d
       ADR r4,x        ; get address for x
       STR r0,[r4]     ; store value of x
after ... ; code after the if statement
```

WRITE AN ARM ASSEMBLY LANGUAGE CODE FOR THE FOLLOWING LOOPS:

```
for (i = 0, f = 0; i < N; i++)
     f = f + c[i] * x[i];
```

```
i = 0;
f = 0;
```

```
while (i < N) {
       f = f + c[i]*x[i];
       i++;
}
```

Here is the code for the loop:

```
     ; loop initiation code
     MOV r0,#0       ; use r0 for i, set to 0
     MOV r8,#0       ; use a separate index for arrays
     ADR r2,N        ; get address for N
     LDR r1,[r2]     ; get value of N for loop termination test
     MOV r2,#0       ; use r2 for f, set to 0
     ADR r3,c        ; load r3 with address of base of c array
     ADR r5,x        ; load r5 with address of base of x array
     ; loop body
loop LDR r4,[r3,r8]    ; get value of c[i]
     LDR r6,[r5,r8]    ; get value of x[i]
     MUL r4,r4,r6    ; compute c[i]*x[i]
     ADD r2,r2,r4    ; add into running sum f
     ; update loop counter and array index
     ADD r8,r8,#4      ; add one word offset to array index
     ADD r0,r0,#1      ; add 1 to i
     ; test for exit
     CMP r0,r1
     BLT loop        ; if i < N, continue loop
loopend...
```

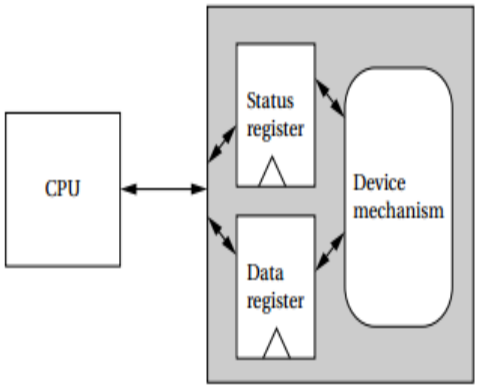
PROGRAMMING THE INPUT AND OUTPUT:

The basic technique for input output programming can be understood relatively independence of the instruction set.

INPUT OUTPUT DEVICES

Input and output devices usually have some Analog or Non electronic component for instance a disk drive has a rotating disk and Analog read/write electronics.

The structure of the typical input output devices and its relationship with the CPU is shown below:



This device has several registers

→Data registers: which holds the value that are treated as data by the device such as the data read or data written by a disk.

→Status register: provides information about the devices option such as whether the current Transaction has been completed or not.

Some registers may be read only such as register that indicates when the device is done. While others may be readable or writable.

INPUT OUTPUT PRIMITIVES:

Microprocessors can provide programming support for input and output in two ways: I/O instructions and memory-mapped I/O. Some architectures, such as the Intel x86, provide special instructions (in and out in the case of the Intel x86) for input and output. These instructions provide a separate address space for I/O devices.

But the most common way to implement I/O is by memory mapping—even CPUs that provide I/O instructions can also implement memory-mapped I/O.

As the name implies, memory mapped I/O provides addresses for the registers in each I/O device. Programs use the CPU's normal read and write instructions to communicate with the devices.

How can we directly write I/O devices in a high-level language like C is explained below:

When we define and use a variable in C, the compiler hides the variable's address from us. But we can use pointers to manipulate addresses of I/O devices.

The traditional names for functions that read and write arbitrary memory locations are peek and poke.

The peek function can be written in c as:

```
int  peek(char *location)  {

return *location; /* de-reference
location pointer */

}
```

The argument to peek is a pointer that is de-referenced by the C * operator to read the location. Thus, to read a device register we can write:

```
#define DEV1 0x1000

. . .

dev_status = peek(DEV1); /* read device
register  */
```

The poke function can be implemented as:

The poke function can be implemented as:

```
void poke(char *location,  char newval) {

(*location) = newval; /* write to
location */

}
```

To write to the status register, we can use the following code:

```
poke(DEV1,8); /*  write  8  to device
register  */
```

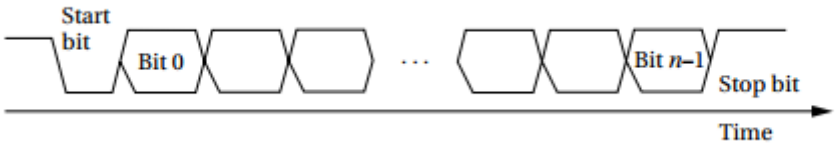
These functions can, of course, be used to read and write arbitrary memory locations, not just devices.

UNIVERSAL ASYNCHRONOUS RECIEVER / TRANSMITTER:

The 8251 UART (Universal Asynchronous Receiver/Transmitter) [Int82] is the original device used for serial communications, such as the serial port connections on PCs. The 8251

was introduced as a stand-alone integrated circuit for early microprocessors. The UART is programmable for a variety of transmission and reception parameters.

Data are transmitted as streams of characters, each of which has the following form:



Every character starts with a start bit (a 0) and a stop bit (a 1). The start bit allows the receiver to recognize the start of a new character; the stop bit ensures that there will be a transition at the start of the stop bit. The data bits are sent as high and low voltages at a uniform rate. That rate is known as the baud rate; the period of one bit is the inverse of the baud rate. Before transmitting or receiving data, the CPU must set the UART's mode registers to correspond to the data line's characteristics. The parameters for the serial port are familiar from the parameters for a serial communications program (such as Kermit):

■The baud rate.

■The number of bits per character (5 through 8).

■Whether parity is to be included and whether it is even or odd.

■ the length of a stop bit (1, 1.5, or 2 bits).

The UART includes one 8-bit register that buffers characters between the UART and the CPU bus. The Transmitter Ready output indicates that the transmitter is ready to accept a data character; the Transmitter Empty signal goes high when the UART has no characters to send. On the receiver side, the Receiver Ready pin goes high when the UART has a character ready to be read by the CPU.

BUSY WAIT INPUT OUTPUT:

Devices are basically slower than the CPU and may require many cycle to compute an operation. If the CPU is performing multiple operations on a single device such as writing several characters to an output device then it must wait for one operation to complete before starting the next one.

INTERRUPTS:

BASICS: busy wait input output is extremely in efficient. The CPU does nothing but test the device status while the input output transaction is in progression.

The CPU could do useful work in parallel with the input output transaction such as:

→Computation as in determining the next output to send to the device on processing the last input received.

→Control of other input output devices.

To allow parallelism, we need to introduce new mechanisms with C to CPU.

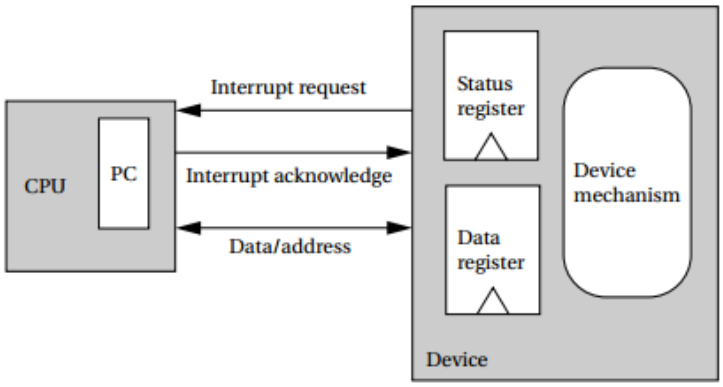
The interrupt mechanism allows the devices to signal the CPU'S to force execution of a particular PIECE OF CODE.

When an interrupt occurs the program counters value is changed to point to an interrupt handler routine (device driver) that takes care of the device writing the next data, reading the data.

Interrupts therefore allow the flow of control in the CPU to change easily between different contexts, such as a foreground computation and multiple I/O devices. The interface between the CPU and I/O device includes the following signals for interrupting:

■ the I/O device asserts the interrupt request signal when it wants service from the CPU.

■ the CPU asserts the interrupt acknowledge signal when it is ready to handle the I/O device's request.



EXAMPLE: copying character from input to output with basic interrupts.

PRIORITIES AND VECTORS:

Providing a practical interrupt system requires having more than a simple interrupt request line. Most systems have more than one I/O device, so there must be some mechanism for allowing multiple devices to interrupt. We also want to have flexibility in the locations of the interrupt handling routines, the addresses for devices, and so on. There are two ways in

which interrupts can be generalized to handle multiple devices and to provide more flexible definitions for the associated hardware and software:

→Interrupt priorities allow the CPU to recognize some interrupts as more important than others, and

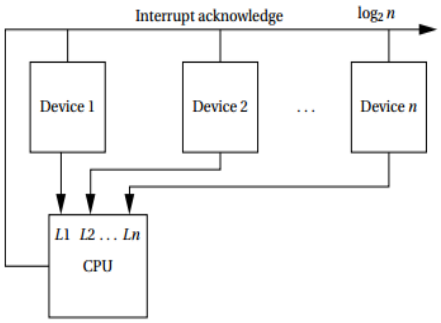
→Interrupt vectors allow the interrupting device to specify its handler.

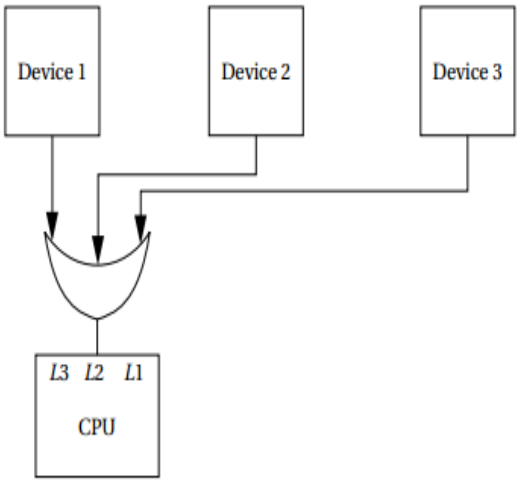
Prioritized interrupts not only allow multiple devices to be connected to the interrupt line but also allow the CPU to ignore less important interrupt requests while it handles more important requests. The CPU provides several different interrupt request signals, shown here as L1, L2, up to Ln.

A device knows that its interrupts requested was accepted by serving its own priority number on the interrupt acknowledge lines.

1) Masking
2) Non maskable interrupt

The below diagrams shows the prioritised device interrupts:





The priority mechanism must ensure that a lower-priority interrupt does not occur when a higher-priority interrupt is being handled. The decision process is known as masking. When an interrupt is acknowledged, the CPU stores in an internal register the priority level of that interrupt. When a subsequent interruptis received, its priority is checked against the priority register; the new request is acknowledged only if it has higher priority than the currently pending interrupt.

When the interrupt handler exits, the priority register must be reset. The need to reset the priority register is one reason why most architectures introduce a specialized instruction to return from interrupts rather than using the standard sub routinereturn instruction.

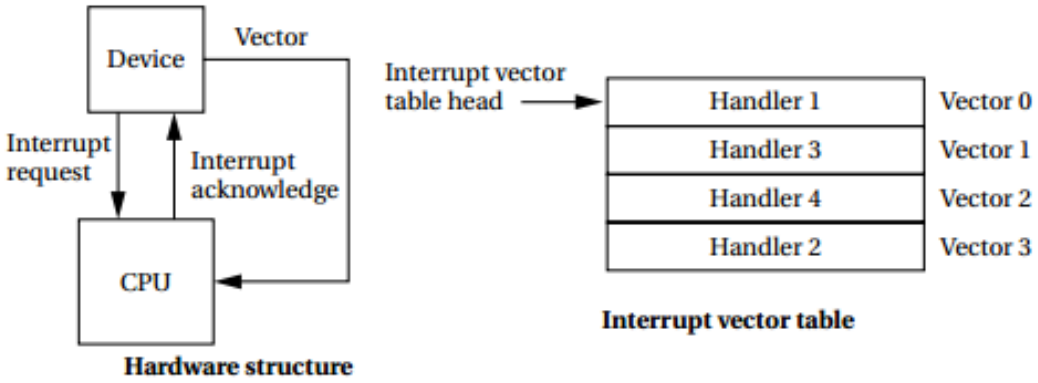
The highest-priority interrupt is normally called the nonmaskable interrupt(NMI). The NMI cannot be turned off and is usually reserved for interrupts caused by power failures—a simple circuit can be used to detect a dangerously low power supply,and the NMI interrupt handler can be used to save critical state in non volatile memory, turn off I/O devices to eliminate spurious device operation during powerdown.

Most CPUs provide a relatively small number of interrupt priority levels, suchas eight. While more priority levels can be added with external logic, they may not be necessary in all cases. When several devices naturally assume the same priority(such as when you have several identical keypads attached to a single CPU), you can combine polling with prioritized interrupts to efficiently handle the devices.

There are two important things to notice about interrupt vector mechanism.

→The device not the cpu, stores its vector number device can be given a new handler by changing the vector number.

→There is no fixed relationship between vector numbers and interrupt handler.



We can now consider a complete interrupt handling process.

→**CPU:** The CPU checks for pending interrupts at the beginning of an instruction.It answers the highest-priority interrupt, which has a higher prioritythan that given in the interrupt priority register.

→**Device**: The device receives the acknowledgment and sends the CPU itsinterrupt vector.

→**CPU**: The CPU looks up the device handler address in the interrupt vectortable using the vector as an index. A subroutine-like mechanism is used tosave the current value of the PC and possibly other internal CPU state, suchas general-purpose registers.

→**SOFTWARE**:The device driver may save additional CPU state. It then performsthe required operations on the device. It then restores any saved state andexecutes the interrupt return instruction.

→**CPU** The interrupt return instruction restores the PC and other automatically saved states to return execution to the code that was interrupted.

Interrupts do not come without a performance penalty. In addition to the execution time required for the code that talks directly to the devices, there is execution time overhead associated with the interrupt mechanisms.

→The interrupt itself has overhead similar to a subroutine call. Because an interrupt causes a change in the program counter, it incurs a branchpenalty. In addition, if the interrupt automatically stores CPU registers, that action requires extra cycles, even if the state is not modified by the interrupt handler.

→ In addition to the branch delay penalty, the interrupt requires extra cycles to acknowledge the interrupt and obtain the vector from the device.

→The interrupt handler will, in general, save and restore CPU registers that were not automatically saved by the interrupt.

→The interrupt return instruction incurs a branch penalty as well as the time required to restore the automatically saved state.

Interrupts in ARM ARM7 supports two types of interrupts: fast interrupt requests (FIQs) and interrupt requests (IRQs). An FIQ takes priority over an IRQ. The interrupt table is always kept in the bottom memory addresses, starting at location 0.

The entries in the table typically contain subroutine calls to the appropriate handler. The ARM7 performs the following steps when responding to an interrupt [ARM99B]:

■ saves the appropriate value of the PC to be used to return.

■ copies the CPSR into a saved program status register (SPSR).

■ forces bits in the CPSR to note the interrupt.

■ forces the PC to the appropriate interrupt vector.

When leaving the interrupt handler, the handler should:

→ restore the proper PC value.

→restore the CPSR from the SPSR.

→clear interrupt disable flags

SUPERVISOR MODE:

Complex systems are often implemented as several programs that communicate with each other. These programs may run under the command of an operating system. In such cases it is often useful to have a supervisor mode provided by the CPU. Normal programs run in user mode. The supervisor mode has privileges that user modes do not. For example, we study memory management systems in Section 3.4.2 that allow the addresses of memory locations to be changed dynam- ically. Control of the memory management unit (MMU) is typically reserved for supervisor mode to avoid the obvious problems that could occur when program bugs cause inadvertent changes in the memory management registers. The ARM instruction that puts the CPU in supervisor mode is called SWI:

SWI CODE_1

SWI causes the CPU to go into supervisor mode and sets the PC to 0x08.

EXCEPTIONS:

An exception is an internally detected error. A simple example is division by zero. One way to handle this problem would be to check every divisor before division to be sure it is not zero, but this would both substantially increase the size of numerical programs and cost a great deal of CPU time evaluating the divisor's value. The CPU can more efficiently check the divisor's value during execution. Since the time at which a zero divisor will be found is not known in advance, this event is similar to an interrupt except that it is generated inside the CPU. The exception mechanism provides a way for the program to react to such unexpected events.

→exceptions must be prioritised because a single operation may generate more than one exception.

→vector provides a way for the user to specify the handler for the exception condition.

TRAPS:

A trap,also known as a software interrupt,is an instruction that explicitly generates an exception condition. The most common use of a trap is to enter supervisor mode. The entry into supervisor mode must be controlled to maintain security—if the interface between user and supervisor mode is improperly designed,a user program may be able to sneak code into the supervisor mode that could be executed to perform harmful operations. The ARM provides the SWI interrupt for software interrupts. This instruction causes the CPU to enter supervisor mode.An opcode is embedded in the instruction that can be read by the handler

CO-PROCESS:

CPU architects often want to provide flexibility in what features are implemented in the CPU. One way to provide such flexibility at the instruction set level is to allow co-processors, which are attached to the CPU and implement some of the instructions.

FOR EXAMPLE: floatingpoint arithmetic was introduced into the Intel architecture by providing separate chips that implemented the floating-point instructions.

→To support co-processors, certain opcodes must be reserved in the instruction set for co-processor operations. Because it executes instructions, a co-processor must be tightly coupled to the CPU.

→When the CPU receives a co-processor instruc- tion, the CPU must activate the coprocessor and pass it the relevant instruction.

→Co-processor instructionscan load and store co-processor registers or can perform internal operations. The CPU can suspend execution to wait for the co-processor instruction to finish.

It can also take a more superscalar approach and continue executing instructions while waiting for the co-processor to finish.

MEMORY SYSTEM MECHANISMS:

Modern microprocessors do more than just read and write a monolithic memory.

→caches to increase the average performance of the memory system. Although memory capacity is increasing steadily, program sizes are increasing as well.

→ Modern microprocessor units (MMUs) perform address translations that provide a larger virtual memory space in a small physical memory.

CACHES:

Caches are widely used to speed up memory system performance.

→designer use this memory for accessing the average performance in ecs.

→A cache is a small, fast memory that holds copies of some of the contents of main memory. Because the cache is fast, it provides higher-speed access for the CPU.

→Caching makes sense when the CPU is using only a relatively small set of memory locations at any one time; the set of active locations is often called the working set .

A cache controller mediates between the CPU and the memory system comprised of the main memory. The cache controller sends a memory request to the cache and main memory. If the requested location is in the cache,the cache controller forwards the location's contents to the CPU and aborts the main memory request; this condition is known as a cache hit . If the location is not in the cache, the controller waits for the value from main memory and forwards it to the CPU; this situation is known as a cache miss.

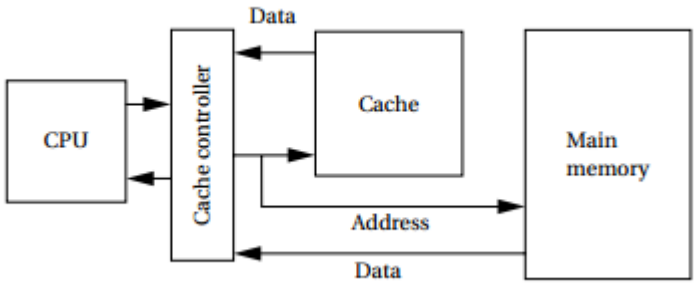


DIAGRAM: represents the cache in the memory system.

We can classify cache misses into several types depending on the situation that generated them:

■ a compulsory miss (also known as a cold miss) occurs the first time a location is used.

■ a capacity miss is caused by a too-large working set.

■ a conflict miss happens when two locations map to the same location in the cache.

We can write some basic formulas for memory system performance.

Let

*h→hit rate(the probability that a given memory location is in the cache).

*1-h→miss rate(probability that the location is not in the cache).

*tcache→access time of the cache.

*tmain→main memory access time.

The average main memory access time is computed by the below formula:

$$t_{av} = bt_{cache} + (1 - b)t_{main}.$$

The probability that location is not in the cache. Two level cache systems, average access time is :

$$t_{av} = b_1 t_{L1} + b_2 t_{L2} + (1 - b_1 - b_2)t_{main}.$$

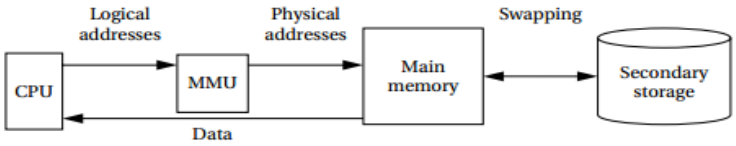
MEMORY MANAGEMENT UNITS AND ADDRESS TRANSLATION:

A MMU translates addresses between the CPU and physical memory. This translation process is often known as memory mapping since addresses are mapped from a logical space into a physical space. MMUs in embedded systems appear primarily in the host processor.

It is helpful to understand the basics of MMUsfor embedded systems complex enough to require them. Many DSPs, including the C55x, do not use MMUs. Since DSPs are used for compute-intensive tasks, they often do not require the hardware assist for logical address spaces.

Early computers used MMUs to compensate for limited address space in their instruction sets. When memory became cheap enough that physical memory could be larger than the address space defined by the instructions, MMUs allowed software to manage multiple programs in a single physical memory,each with its own address space.

Because modern CPUs typically do not have this limitation, MMUs are used to provide virtual addressing .The diagram below shows the virtual addressing.



The MMU accepts logical addresses from the CPU. Logical addresses refer to the program's abstract address space but do not correspond to actual RAM locations.The MMU translates them from tables to physical addresses that do correspond to RAM. By changing the MMU's tables, you can change the physical location at which the program resides without modifying the program's code or data. (We must, of course, move the program in main memory to correspond to the memory mapping change.)

Furthermore, if we add a secondary storage unit such as flash or a disk, we can eliminate parts of the program from main memory. In a virtual memory system, the MMU keeps track of which logical addresses are actually resident in main memory; those that do not reside in main memory are kept on thesecondary storage device.

When the CPU requests an address that is not in main memory, the MMU generates an exception called a page fault . The handler for this exception executes code that reads the requested location from the secondary storage device into main memory. The program that generated the page fault is restarted by the handler only after

■ the required memory has been read back into main memory.

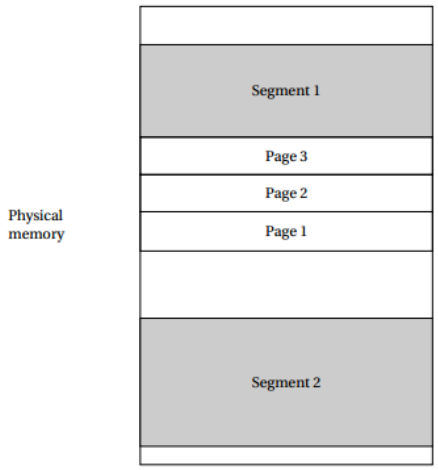
■ the MMU's tables have been updated to reflect the changes.

Of course, loading a location into main memory will usually require throwing something out of main memory. The displaced memory is copied into secondary storage before the requested location is read in. As with caches, LRU is a good replacement policy.

There are two styles of address translation: segmented and paged . Each has advantages and the two can be combined to form a segmented, paged addressing scheme. Segmenting is designed to support a large, arbi- trarily sized region of memory, while pages describe small, equally sized regions.

A segment is usually described by its start address and size, allowing different segments to be of different sizes. Pages are of uniform size, which simplifies the hardware required for address translation. A segmented, paged scheme is created by dividing each segment into pages and using two steps for address translation.

Paging introduces the possibility of fragmentation as program pages are scattered around physical memory.Figures below show segmentation and pages:



CPU PERFORMANCE:

The ARM7 has a three-stage pipeline:

■ Fetch the instruction is fetched from memory.

■ Decode the instruction's opcode and operands are decoded to determine what function to perform.

■ Execute the decoded instruction is executed. Each of these operations requires one clock cycle for typical instructions. Thus, a normal instruction requires three clock cycles to completely execute, known as the latency of instruction execution. In other words, the pipeline has a throughput of one instruction per cycle. The C55x includes a seven-stage pipeline [Tex00B]:

1. Fetch.

2. Decode.

3. Address computes data and branch addresses.

4. Access 1 reads data.

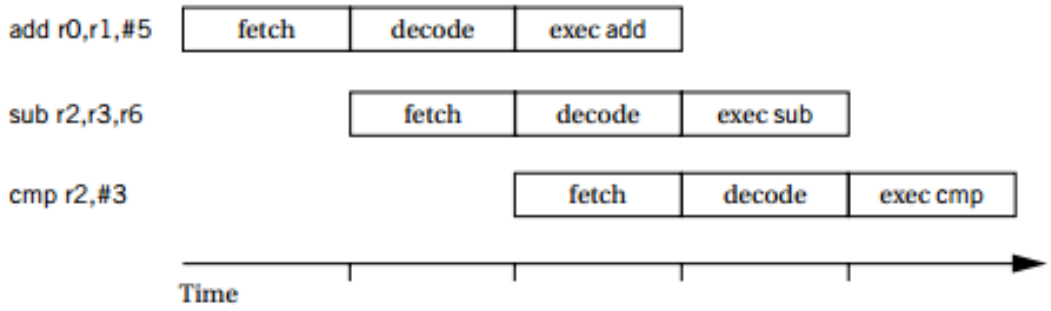
5. Access 2 finishes data read.

6. Read stage puts operands onto internal busses.

7. Execute performs operations.

RISC machines are designed to keep the pipeline busy. CISC machines may display a wide variation in instruction timing. Pipelined RISC machines typically have more regular timing characteristics—most instructions that do not have pipeline hazards display the same latency

FIGURE: below shows the pipelined executions of ARM instructions:



CPU POWER CONSUMPTION:

Power consumption is, in some situations, as important as execution time.

→First, it is important to distinguish between energy and power. Power is, of course, energy consumption per unit time.

→Heat generation depends on power consumption. Battery life, on the other hand, most directly depends on energy consumption.

→The high-level power consumption characteristics of CPUs and other system components are derived from the circuits used to build those components.

→virtually all digital systems are built with complementary metal oxide semiconductor (CMOS) circuitry. The basic sources of CMOS power consumption are easily identified and briefly described below.

■**Voltage drops**: The dynamic power consumption of a CMOS circuit is proportional to the square of the power supply voltage (V2). Therefore, by reducing the power supply voltage to the lowest level that provides the required performance, we can significantly reduce power consumption. We also may be able to add parallel hardware and even further reduce the power supply voltage while maintaining required performance .

■**Toggling :** A CMOS circuit uses most of its power when it is changing its output value. This provides two ways to reduce power consumption. By reducing the speed at which the circuit operates, we can reduce its power consumption (although not the total energy required for the operation, since the result is available later). We can actually reduce energy consumption by eliminating unnecessary changes to the inputs of a CMOS circuit—eliminating unnecessary glitches at the circuit outputs eliminates unnecessary power consumption.

■**Leakage**: Even when a CMOS circuit is not active, some charge leaks out of the circuit's nodes through the substrate. The only way to eliminate leak- age current is to remove the power supply. Completely disconnecting the power supply eliminates power consumption, but it usually takes a significant amount of time to reconnect the system to the power supply and reinitialize its internal state so that it once again.

There are two types of power management features provided by CPUs.

→**A static power management mechanism** is invoked by the user but does not otherwise depend on CPU activities.

AN EXAMPLE: of a static mechanism is a power- down mode intended to save energy. This mode provides a high-level way to reduce unnecessary power consumption. The mode is typically entered with an instruction. If the mode stops the interpretation of instructions, then it clearly cannot be exited by execution of another instruction. Power-down modes typically end upon receipt of an interrupt or other event.

→**A dynamic power management mechanism** takes actions to control power based upon the dynamic activity in the CPU.

FOR EXAMPLE: the CPU may turn off certain sections of the CPU when the instructions being executed do not need them.

SUMMARY:

Numerous mechanisms must be used to implement complete computer systems.

■ Two major styles of I/O are polled and interrupt driven.

■ Interrupts may be vectorized and prioritized.

■ Supervisor mode helps protect the computer from program errors and provides a mechanism for controlling multiple programs.

■ An exception is an internal error; a trap or software interrupt is explicitly generated by an instruction. Both are handled similarly to interrupts.

■ A cache provides fast storage for a small number of main memory locations. Caches may be direct mapped or set associative.

■ A memory management unit translates addresses from logical to physical addresses.

■ Co-processors provide a way to optionally implement certain instructions in hardware.

■ Program performance can be influenced by pipelining, superscalar execution, and the cache. Of these, the cache introduces the most variability into instruction execution time.

■ CPUs may provide static (independent of program behavior) or dynamic (influenced by currently executing instructions) methods for managing power consumption.

Unit – 5

Real Time Operating System RTOS Based Design

The process & the operating system OS . These two abstractions lets us switch the state of the process between multiple tasks. The process clearly defines the status of an executing between programs while the OS provides the mechanism for switching execution between the processes.

Operating System Basics

- Bridges between user application/tasks & system resources
- The OS manages the system resources & make them available to the user application/task.
- A computing system is collection of different I/O sub-system working & storage memory.

PrimaryFunction Of OS

- Make the system convenient to use
- Organizes & manage the system resources efficiently & correctly

The Kernel

The kernel is core of the OS & is responsible for managing the system & the communication among the hardware & the system services. Kernel acts as the abstraction layer between system resources & user application. Kernel contains a set of system libraries of services. For general purpose OS the kernel contains different services for handling the management.

1. Process Management :

- Managing the process/tasks
- Setup the memory space for process
- Load program/code into space(memory)
- Scheduling & managing the execution of the process
- Setting up & managing the PCB
- Inter process communication & system synchronization process termination & deletion

2. Primary Memory Management

- It is volatile memory
- The MMU of kernel is responsible for
  - Keeping track of which part of memory area is correctly used by which process

- Allocating & deallocating the memory spaces

3. File System Management

- File is a collection of related information
- A file could be a program, tent files, image file, word documents, audio/video files etc.
- The file system management service of kernel is responsible for
  - The creation & deletion of files
  - Creation , deletion & alteration directly
  - Saving the file in secondary storage memory
  - Providing automatic allocation of spaces
  - Providing a flexible naming conversion for the files

4. I/O System(Device) Management

- Kernel is responsible for routing the I/O request coming from different user applications
- Direct access of I/O devices are not allowed , we can access the I/O devices through the API imposed by kernel
- Dynamically update the available devices
- Device manager of the kernel is responsible for handling I/O device related operations
- The kernel talks to the I/O device through the device driver , this is responsible for
  - Loading & unloading of device drivers
  - Exchanging the informations

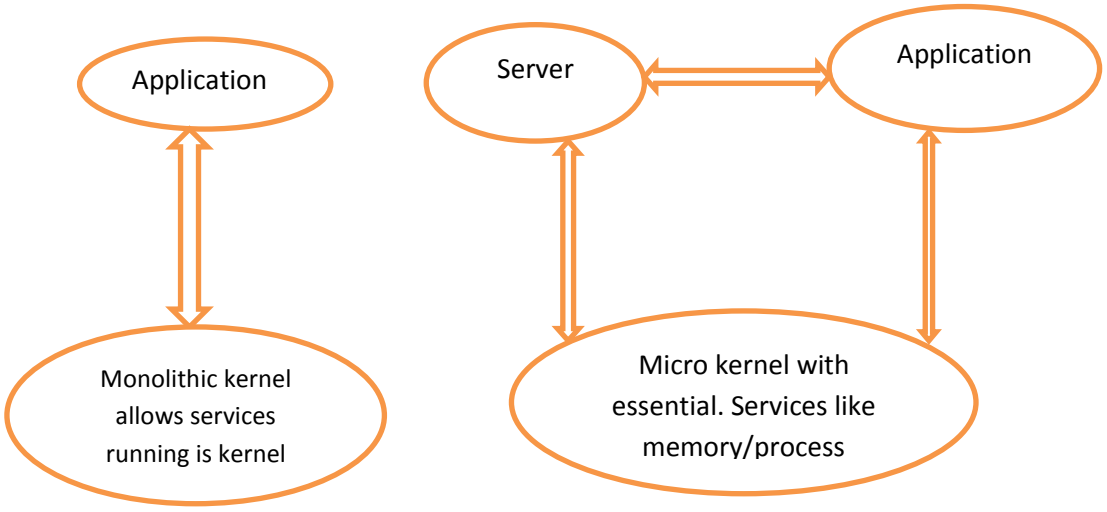
5. Secondary Storage Management

- The secondary storage management deals with secondary storage memory device , if any connected to the system
- The main memory is volatile
- The secondary storage management services of kernel deals with
  - Disk storage allocation
  - Disk scheduling
  - Free disk space management

Types of Kernel:

Based on kernel design, kernel can be classified into Monolithic & Micro

- In Monolithic kernel architecture all the kernel service run in the kernel space – means all kernel module run with same memory space under single kernel thread
- The drawback of Monolithic kernel is that any error or failure in any of the kernel module i.e. leads to the crashing of entire kernel application. Ex : LINUX,SOLARIS,MS-DOS



Monolithic kernel model **Micro kernel**

Micro kernel: design incorporates only the essential set of OS services into the kernel. The rest of the OS services are implanted in programs known as "servers" which runs in user space.

The essential services of the Micro kernel are

- Memory management
- Process management
- Timer system
- Interrupt handler

Ex. OS , MACH , ONX , MINIX3

Benefits of Micro kernel

1. Robust
2. Configurability

Types of Operating System

Depending on the type of kernel & kernel services purpose & type of in computing system OS are classified into two types

1. General Purpose Operating System[GPOS]
2. Real Time Operating System[RTOS]

1. General Purpose Operating System[GPOS]

The OS which are deployed in general computing systems are referred as general purpose OS.

- The kernel of such OS is more generalized & it contains all kinds of services required for executing generic applications
- GPOS are non-deterministic in behavior

Ex: PC/Desktop system, windows XP & MS-DOS

2. Real Time Operating System [RTOS]

There is no universal definition available for the term RTOS.

- A RTOS is an OS intended to serve real-time application requests. It must be able to process the data as it comes in typically without buffering delays
- Processing time requirements are measured in tenth of seconds. Hard RTOS has less jitter than soft RTOS.

[Jitter: Variation in time between packet arriving with time congestion & time.]

- Real-time implies deterministic timing behavior – means the OS services consumes only known & expected amount of time regardless the no of services.
- RTOS decides which application should run in which order & how much time needs to be allocated for each applications.
  Ex: Windows CE, UNIX,VxWorks,Micros/OS-II

Real Time Kernel:

The RTOS is referred to as Real Time kernel in complement to conventional OS kernel. The kernel is highly specialized & it contains only the minimal set of services required for running the user application/ tasks.

The Basic Functions Of Real Time Kernel:

1. Task/Process Management
2. Task/Process Scheduling
3. Task/Process Synchronization
4. Error/Exception Handling
5. Memory Management
6. Interrupt Handling
7. Time Management

Task/Process Management: Deals with setting up the memory space for the tasks, loading the tasks code into memory space, allocating system resources, setting up a Task Control Block [TCB] for the task & task/process termination/deletion.

TCB: Task Control Block is used for holding the information corresponding to a task. The TCB contains the following set of information

- Task ID: Task identification number
- Task State: The current state of the task
- Task Type: Indicates what is the type of this task, the task can be hard real time or soft real time or background task
- Task Priority: Ex task priority=1 for task with priority =1
- Task Context Pointer: context pointer, pointer for saving context
- Task Memory Pointer: Pointers to the code memory, data memory & stack memory for the task
- Task System Resource Pointers: Pointer to system resource used by the task
- Task Pointers: Pointer to other TCB's
- Other Parameters: Other relevant task parameters

The TCB parameters vary across different kernels based on the task management Implementation

- Creates TCB for a task on creating task
- Delete/Remove the TCB when the task is terminated or deleted
- Reads the TCB to get the state of a task
- Updates the TCB with updated parameters on need basics
- Modify the TCB to change the priority of task dynamically

Task/Process Scheduling:Sharing the CPU among various tasks/process. A kernel applications called scheduler, handles the task scheduling. Scheduler is nothing but an algorithm implemented all to , which performs the efficient & optimal scheduling of task to provide a deterministic behavior.

Task/Process Synchronization: deals with system concurrent accessing a resource which is shared access multiple tasks & communication between various tasks.

Error/Exception Handling: Deals with registering& handling the errors occurred during the execution of tasks. Ex: Insufficient memory, time outs, dead locks, dead line missing, bus error, divide by zero, unknown instruction execution. Errors & exceptions can happen at two levels of services * Kernel Level Service * At Task Level

Memory Management: RTOS makes use of 'Block Based Memory' allocation techniques instead of the usual dynamic memory allocation technique used by GPOS. RTOS kernel uses blocks of fixed size dynamic memory & block is allocated for a task on a need of basis. A few RTOS kernel implements Virtual Memory concepts avoid the garbage collection overhead.

Interrupt Handler: Deals with the handling of several of interrupts. Interrupts provide Real Time behavior to systems. Interrupts inform the processor that an external device or an

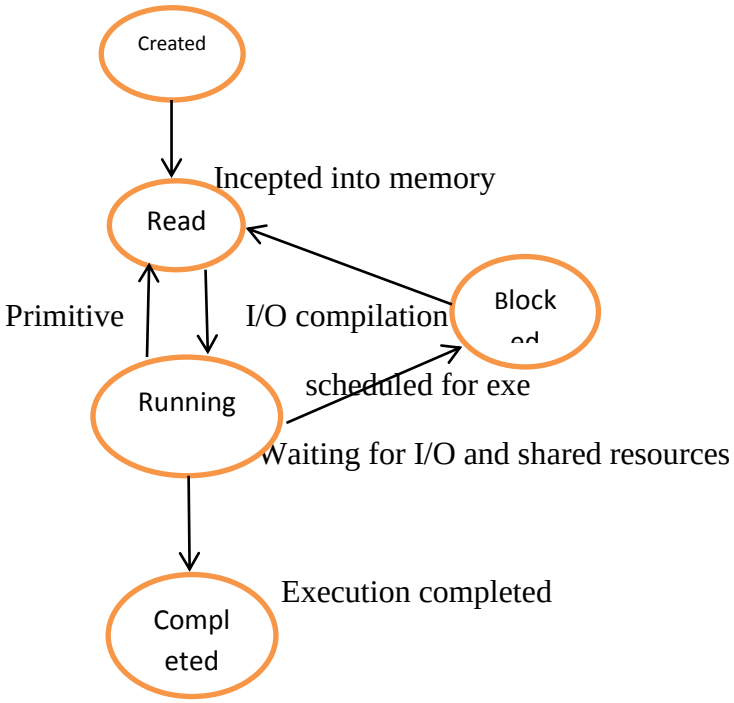
associated task required immediate attention of the CPU. Interrupts can be either Synchronous Asynchronous. Interrupts which occurs in synch with the currently executing task is known as synchronous interrupts. Usually the system interrupts fall under the synchronous category Ex Divide by zero, memory segmentation error. A synchronous interrupts are those which occurs at any point of execution of any task & are not in sync with currently executing tasks.

Task, Process & Threads

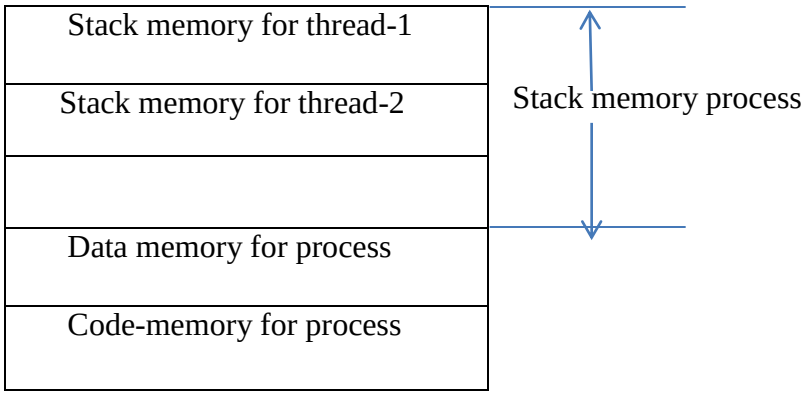
Task: Refers to something that needs to be done. The tasks refers can be one assigned by manages or the one assigned by only one of the processor family needs. In addition we have an order of priority & scheduling timeline for executing these tasks. In OS context, a task is defined as the program in execution & related information maintained by the OS. Task is also known as "JOB" in the OS context.

Process: A process is a program as part of the program, in execution process is also known as instance of the program in execution, multiple instance of the same program can execute simultaneously. Process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process, I/O devices for in function exchange etc.

Process States & State Transition:



The structure Of Process



Memory organization of a process and its associated threads

Run: A task enters this state as it starts executing on the process

Ready state of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task

Blocked a tasks enters this state when it executes synchronization primitive to wait for an event

Ex:-a wait primitive on a semaphore this case the task is inserted in a queue associated with the semaphore.

Created (idle) a periodic job enters this state when it computes its execution and has to wait for the beginning of the next period

Process management:

Process management deals with the creating a process, setting up the memory space for the process, loading the process code with the memory space, allocating the system resources. Setting up a process control block (PCB) for the process and process termination/detection.

Multiprocess multitasking

The terms multiprocessing and multitasking are causing and sounds a like.

Multiprocessing describes the ability to execute the multiple processes simultaneously

Systems which are capable of performing multiprocessing are known as multiprocessor systems.

Multiprocessor systems process multiple CPU's and can execute multiple processes simultaneously

The ability of an operating system to hold multiple processes in memory and switch the process is known as multitasking.

->multitasking creates the illusion of multiple tasks executing in parallel.

->involves the switching of CPU from executing one task to another

Process is considered as a virtual processor waiting its turn to have its properties switched into the physical processor

In a multitasking environment when task/process switching happens the virtual process gets its properties converted into that of the physical processor

The switching of the virtual processor to physical processor is controlled by the scheduling of the OS kernel

Whenever a CPU switching happens the current content execution should be saved to retrieve it at a later part of time when the CPU executes the process, which is interrupted currently due to execution switching.

The context saving and retrieval is essential for resuming a process exactly from the point is where it is interrupted due to CPU switching

The act of switching CPU among the process or changing the current execution context is known as "content switching"

The act of saving the current content which contains the content details for the currently running process at the time of CPU switching is known as "content saving"

The process of retrieving the saved content details for a process which is going to be executed due to CPU switching is known as content retrieval

Multitasking involves "context switching" context saving and context retrieval

Types of multitasking

Co-operative multitasking

Primitive multitasking

Non-primitive multitasking

Co-operative multitasking in which a task/process gets chance to execute only when currently executing task/process voluntarily relinquishes the CPU.That any task/process can hold the CPU as much time as it waits. Since this type of implementation involves the merging of the tasks each other for getting the CPU time for executing.

Primitive multitasking

It ensures that every task/process gets a chance to execute as the name indicates in primitive multitasking the currently running task/process is preempted to give a chance to other tasks/process to execute

Non-preemptive multitasking

The process/task which is currently given the CPU time is allowed to execute until it terminates of enter the blocked/wait state.

The schedule of scheduling criteria/algorithm considers the following factor

CPU utilization:-scheduling algorithm should always make the CPU utilization high

Throughput:-this is given an indication of time taken by a process executed per unit of time

Turnaround time:-it is the amount of time taken by a process for computing its execution it includes the time spent by the process for waiting for the main memory

Thread:

A thread is the primitive that can be executed code.

A thread is a single sequential flow of control within process. Thread is also known as light weight process. A process can have many threads of execution. Different threads which are part of a process share the same adress space

Win 32, java.threads, posix threads

Task scheduling:

Multitasking envoles the execution switching among the different tasks. There should be same mechanism in place to show the CPU among the different taks and to decide which process/task is to be executed at a given point of time.

Determining which task/process is to be executed at a give point of time is known as task/process scheduling.

Task scheduling froms the basis of multitasking

Scheduling policies froms the guidelines for determining which task is to be executed when.

The work of choosing the order of running process is known as scheduling.

A scheduling policy defines how processes are selected for promotion from the ready state to the running sate.

The process scheduling decision may take place when a process

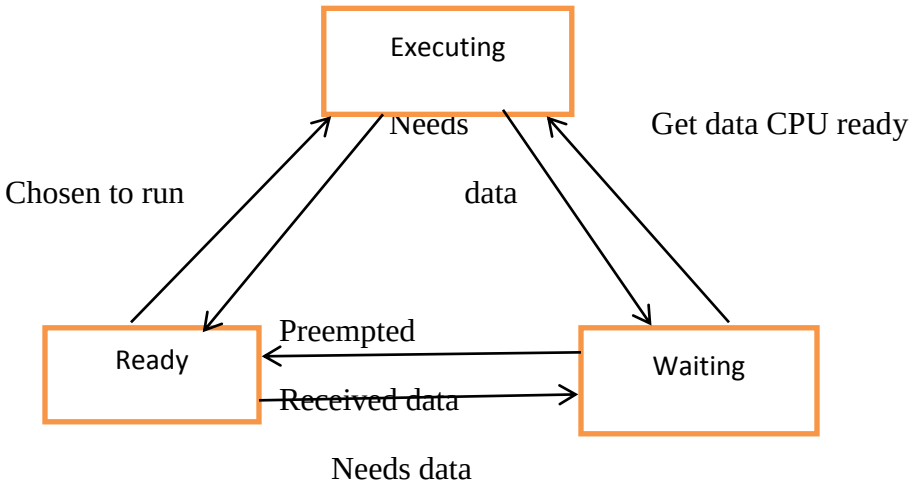
Switches its states

Ready state from Running state

Blocked/wait sate from Running stae

Ready state from Blocked/wait state

Completed (executed) state



Scenario 1 is premitive:a process swithches to ready state from the running state

Scenario 2 is scheduling under can be either preemptive or non-preemptive

When a high priority process in the blocked/wait state complete it's I/O and switches to ready state. The scheduler picks it for execution if the scheduling policy used is priority based preemptive.

UNIT-6

RTOS-Based Design-2

Inter process Communication

In general, a process can send a communication in one of two ways: ***blocking*** or ***nonblocking*** . After sending a blocking communication, the process goes into the waiting state until it receives a response.

Nonblocking communication allows the process to continue execution after sending the communication. Both types of communication are useful.

There are two major styles of interprocess communication: ***shared memory*** and ***message passing*** . The two are logically equivalent—given one, you can build an interface that implements the other. However, some programs may be easier to write using one rather than the other. In addition, the hardware platform may make one easier to implement or more efficient than the other.

Message Passing

Message passing communication complements the shared memory model. As shown in Figure 6.1, each communicating entity has its own message send/receive unit. The message is

not stored on the communications link, but rather at the senders/ receivers at the end points. In contrast, shared memory communication can be seen as a memory block used as a communication device, in which all the data are stored in the communication link/memory.

generated by a process and transmitted to another process

by the operating system.

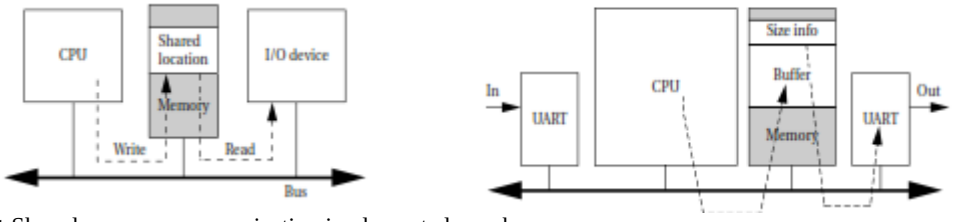


Fig 6.1 Shared memory communication implemented on a bus.

A UML signal is actually a generalization of the Unix signal. While a Unix signal carries no parameters other than a condition code, a UML signal is an object. As such, it can carry

parameters as object attributes. Figure 6.16 shows the use of a signal in UML. The *sigbehavior*( ) behavior of the class is responsible for throwing the signal, as indicated by *send* . The signal object is indicated by the *signal* stereotype.

EVALUATING OPERATING SYSTEM PERFORMANCE

The scheduling policy does not tell us all that we would like to know about the performance of a real system running processes. Our analysis of scheduling policies makes some

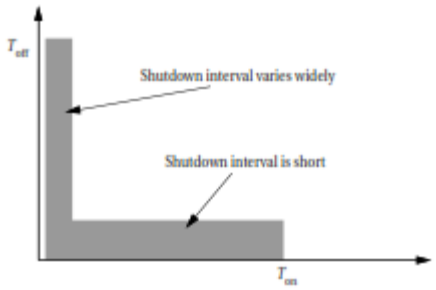
simplifying assumptions:

1.We have assumed that context switches require zero time. Although it is often reasonable to neglect context switch time when it is much smaller than the process execution time, context switching can add significant delay in some cases.

2.We have assumed that we know the execution time of the processes.Tthe program time is not a single number, but can be bounded by worst case And best-case execution times.
3.We probably determined worst-case or best-case times for the processes in isolation. But,in fact,they interact with each other in the cache. Cache conflicts among processes can manage the system's power consumption. A ***power management policy*** [Ben00] is a strategy for determining
4.when to perform certain power management operations. A power management policy in general examines the state of the system to determine when to take actions. However, the overall strategy embodied in the policy should be designed based on the characteristics of the static and dynamic power management mechanisms.
Going into a low-power mode takes time; generally, the more that is shut off, the longer the delay incurred during restart. Because power-down and power-up are not free, modes should be changed carefully. Determining when to switch into and out of a power-up mode requires an analysis of the overall system activity.

5. Avoiding a power-down mode can cost unnecessary power.

6.Powering down too soon can cause severe performance penalties.



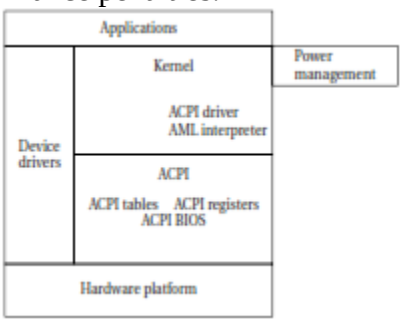


Fig 6.2 An L-shaped usage distribution. Fig.6.3The advanced configuration and power interface and its relationship to a complete system.

Re-entering run mode typically costs a considerable amount of time.
A straightforward method is to power up the system when a request is received. This works as long as the delay in handling the request is acceptable. A more sophisticated technique is ***predictive shutdown***. The goal is to predict when the next request will be made and to start the system just before that time, sav- ing the requestor the start-up time.
In general, predictive shutdown techniques are probabilistic—they make guesses about activity patterns down system waits for a period *T*off before returning to the power-on mode. The choice of *T* off and *T* on must be determined by experimentation. Srivastava and Eustace [Sri94] found one useful rule for graphics terminals. They plotted the observed idle time (*T* off *)* of a graphics terminal versus the immediately preceding active time (*T*on *)*. The result was an L-shaped distribution as illustrated in Figure 6.17. In this distribution, the idle period after a long active period is usually very short, and the length of the idle period after a short active period is uniformly distributed. Based on this distribution, they

proposed a shut down threshold that depended on the length of the last active period—they shut down when the active period length was below a threshold, putting the system in the vertical portion of the *L* distribution.

The ***Advanced Configuration and Power Interface (ACPI)***
is an open indus- try standard for power management services. It is designed to be compatible with a wide variety of OSs. It was targeted initially to PCs. The role of ACPI in the system is illustrated in Figure 6.3. ACPI provides some basic power management facilities and abstracts the hardware layer, the OS has its own power management module that determines the policy, and the OS then uses ACPI to send the required controls to the hardware and to observe the
hardware's state as input to the power manager.
ACPI supports the following five basic global power states:
i. G3, the mechanical off state, in which the system consumes no power.

ii. G2, the soft off state, which requires a full OS reboot to restore the machine to working condition. This state has four analog tape. To make life more interesting, we use a simple algorithm to compress the voice data so that we can make more efficient use of the limited amount of available memory.

Theory of Operation and Requirements

In addition to studying the compression algorithm, we also need to learn a little about the operation of telephone systems.

The compression scheme we will use is known as ***adaptive differential pulse code modulation (ADPCM)***. Despite the long name, the technique is relatively simple but can yield 2 compression ratios on voice data.
The ADPCM coding scheme is illustrated in Unlike traditional sampling, in which each sample shows the magnitude of the signal at a particular time, ADPCM encodes changes in the signal. The samples are expressed in a ***coding alphabet*** , whose values are in a relative range that
spans both negative and positive values. In this case, the value range is { 3, 2, 1, 1, 2, 3}.
Each sample is used to predict the value of the signal at the current instant from the previous value. At each point in time, the sample is chosen such that the error between the predicted value and the actual signal value is minimized.
An ADPCM compression system, including an encoder and decoder, is shown in Figure 6.20. The encoder is more complex, but both the encoder and decoder use an integrator to reconstruct the waveform from the samples. The integrator simply computes a running sum of the history of the samples; because the samples are differential, integration a ringing signal to the telephone when a call is waiting. The ringing signal is in fact a 90 V RMS sinusoid, but we can use analog circuitry to produce 0 for no ringing and 1 for ringing.

Off-hook: The telephone industry term for answering a call is going ***off- hook***; the technical term for hanging up is going ***on-hook***. (This creates some initial confusion since *off-hook* means the telephone is active and *on-hook* means it is not in use, but the

terminology starts to make sense after a few uses.) Our interface will send a digital signal to take the phone line off-hook, which will cause analog circuitry to make the necessary connection so that voice data can be sent and received during the call.

We can now write the requirements for the answering machine. We will assume that the interface is not to the actual phone line but to some circuitry that provides voice samples, off-hook commands, and so on. Such circuitry will let us test our system with a telephone line simulator and then build the analog circuitry necessary to connect to a real phone line. We will use the term ***outgoing message (OGM)*** to refer to the message recorded by the owner of the machine and played at the start of every phone call.

Name Digital telephone answering machine

Telephone answering machine with digital memory, using speech compression.

Inputs *Telephone:* voice samples, ring indicator. back.Performance Should be able to record about 30 min of total voice, including incoming and OGMs.

Voice data are sampled at the standard telephone rate of 8kHz.

Specification

The class diagram for the answering machine. In addition to the classes that perform the major functions, we also use classes to describe the incoming and OGMs. As seen below, these classes are related.

The definitions of the physical interface classes are shown in Figure 6.22. The buttons and lights simply provide attributes for their input and output values. The phone line, microphone, and speaker are given behaviors that let ussample their current values.

The message classes are defined in Figure 6.23. Since incoming and OGM types share many characteristics, we derive both from a more fundamental message type. The major operational classes—*Controls*, *Record*, and *Playback*—are defined in Figure 6.4. The *Controls* class provides an *operate*( ) behavior that oversees the userlevel operations.

Fig.6.4 operational classes for the answering machine.

The *Record* and *Playback* classes provide behaviors that handle writing and reading sample sequences.

The state diagram for the *Controls activate* behavior is shown in. Most of the user activities are relatively straightforward. The most complex is answering an incoming call. Want to be sure that a single depression of a button causes the required action to be taken exactly once; this requires edge detection on the button signal. on-hook commands.

¦ The ***telephone input and output modules*** handle receiving samples from and sending samples to the telephone line.

¦ The ***compression module*** compresses data and stores it in memory.

¦ The ***decompression module*** uncompresses data and sends it to the speaker module.

We can determine the execution model for these modules based on the rates at which they must work and the ways in which they communicate.

¦ The front panel and telephone line modules must regularly test the buttons and phone line, but this can be done at a fairly low rate. As seen below, they can therefore run as polled processes in the software's main loop.

```
while (TRUE)
 { check_phone_line();
run_front_panel();
}
```

¦ The speaker and phone input and output modules must run at higher, regular rates and are natural candidates for interrupt Performance analysis is important in this case because we want to ensure that we don't spend so much time compressing that we miss voice samples. In a real consumer product, we would carefully design the code so that we could use the slowest, cheapest possible CPU that would still perform the required processing in the available time
between samples. In this case, we will choose the microprocessor in advance for simplicity and simply ensure that all the deadlines are met.
An important class of problems that should be adequately tested is memory overflow. The system can run out of memory at any time, not just between messages. The modules should be tested to ensure that they do reasonable things when all the available memory is used up.

System Integration and Testing
We can test partial integrations of the software on our host platform. Final testing with real voice data must wait until the application is moved to the target platform. Testing your system by connecting it directly to the phone line is not a very good idea. In the United States, the Federal Communications Commission regulates equipment connected to phone lines. Beyond legal problems, a bad circuit can dam- age the phone line and incur the wrath of your service provider. The required analog circuitry also requires some amount of tuning, and you need a second telephone line to generate phone calls for tests. You can build a telephone line simulator to test the hardware independently of a real telephone line.

UNIT-8

EMBEDDED STSTEM DEVELOPMENT ENVIRONMENT

OBJECTIVES: To learn about

- The different entities of embedded system development environment.
- Integrated development environments for embedded firmware development and debugging.
- The different IDE for firmware development.
- The different types of files
- The disassembler and de-compiler and their role in embedded firmware development.
- Simulators, Emulators.

The embedded system development environment consists of development computer(pc) or host which acts as heart of the development environment

Integrated development environment (IDE) tool for embedded firmware development and debugging.

Electronic design automation (EDA) tool for embedded hardware design.

An emulator hardware for debugging the target board. Signal sources (like function generator) for simulating the input to the target board. target hardware debugging tools digital CRO, multi-meter, logic analyzer and target hardware.

The Integrated Development Environment IDE

In embedded system development context,(IDE) stands for an integrated environment for developing, debugging the target processor specific embedded firmware.

IDE is a software package that bundles, a text editor (source code editor), cross compiler (for cross platform development and compiler for same platform development).

Linker and debugger

Some IDE's may provide interface to target board emulators, target processors/controllers, flash memory programmer.

IDE's can either be command line based or GUI based, command line based IDE's may include little or less GUI support.

Ex: command line based interface of IDE is the old version of TURBO C IDE. For developing applications in C/C++ for X86 processor on windows platform

GUI based IDE provides visual development environment with mouse click support for each action. Such IDE's are generally known as virtual IDE's .visual IDE's are very helpful in firmware development.

Example for virtual IDE, Microsoft virtual studio for developing visual c++& visual basics programs. others are xlet Beans & Eclipse.

TYPES OF FILES GENERATED ON CROSS COMPILATION

Cross Compilations: is the process of converting a source code written inhigh level language(embedded C) to target processor or controller understandable machine code(ex : ARM processor or 8051).

The conversion of code is done by software running on a processor/controller (86 processor based pc) which is different from the target processor. The software performing this operation is referred as the cross compiler.

Cross assembly is similar to cross compiling. The only difference is that the code written in target processor controller assembly code is converted with its corresponding machine code.

Cross compilation / cross assembling is carried out in different steps and the process generates various types of intermediate files

The applications converting assembly instructions to target processor or controller specific code is known as cross assembler.

The various files generated during the cross compilation or cross assembly process are: .lst file , hex file(.hex) pre-processor output file map file, object file(.obj)

LIST FILE(.lst file)

Listing file is generated during the cross compilation process and it contains abundance of information about the cross compilation process.

Cross compiler details: compiler version number, source file name, date, time, and page number page header formatted source text(C code).

Command line that was used to invoke the compiler .source code listing outputs the line number as well as the source code on that line

ASSEMBLY LISTING

Assembly listing contains the assembly code generated by the cross compiler for the C source code.

SYMBOL LISTING

Contains symbolic information about the various symbols present in the cross compiled source file symbol listing contains the sections. Symbol name ,(name) classification structure, types, static, public, auto, extern memory space(MSPACE) data type(TYPE) offset is size in bytes.

NAME	CLASS	MSPACE	TYPE	OFFSET SIZE
Main	PUBLIC	CODE	-------	2
Printf	EXTERN	CODE	0000H	----------

MODULE INFORMATION : the module information provides the size of initialized and uninitialized memory areas defined by the source file

MODULE INFORMATION	STATIC	OVER LAYABLE
Code size	9	---
Constant size	14	---
Xdata side	---	---

WARNING AND ERRORS : warning and errors section of list file records the errors encountered o any statement that may create issue in applications(warning) during cross compilation

PREPRPOCESSORS OUTPUT FILE: contain the pre-processor output for the pre-processor instructions used in the source file pre-processor output files used for verifying the operation of macros and conditional pre-processor directions

The pre-processor output file is invalid C source file. File extension of pre-processor output file is cross compilation dependent.

OBJECT FILE(.OBJ FILE)

Cross compiling /assembling each source module(C/assembly)

Converts the various embedded C/assembly instructions and other directions present in the module to an object (.obj) file the object file is specially formatted file with data records for symbolic information object code, debugging information object code, debugging information library references.

Types of data supported by typical object file format

Header (descriptive and control information)

Text segment (executable code)

Data segment (static code)

External definition and reference for linking

Relocation information

Dynamic linking information

Debugging information to help synchronise server lines with object code

MAP FILE(.map file)

The cross compiler converts each source code module into a relocatable object(OBJ) file cross compiling each source code module generates its own first file.

The object file created are re-locatable codes i.e the location in the code memory is not fined.

It's the responsibility of a linker to link all these object files.

The locator is responsible for locating absolute address to each module in the code memory

DISASSEMBLER / DECOMPILER

Disassembler is a utility program which converts machine codes into target processor specific assembly code instructions

The process of converting machine code into assembly code known as disassembly.

De-compiler is a utility program for translating machine codes into corresponding high level language instructions

De-compiler performs the reverse operations of compilers/cross compilers disassemble /de-compilers are deployed in reverse engineering. Reverse engineering is the process of relating the technology behind the working of a product reverse engineering is embedded product development is employed to find out the secret behind the working products.

Disassembler /de-compilers are powerful tools for analysing the presence of malicious code in an executable way. this is available as either freeware tools readily available for free download from internet.

The information generally held by map files is listed below

PAGE HEADER : indicates the linker version ,date, time, page number,BL51 linker/locater v3.62 02/29/2004 9:59:51 page 1

COMMAND LINE: represents the entire command line that was invoked by the linker

CPU DETAILS

SIMULATOR, EMULATORS AND DEBUGGING

Simulator is a s/w tool used for simulating the various conditions for checking the functionality of the application firmware.

The IDE itself will be providing simulator support and they help in debugging the firmware for checking its requires functionality.

Simulator refers to soft model(GUI model)of the embedded product.

Simulators simulate the target hardware and the firmware execution can be inspected using the simulators.

The features of simulator based debugging are parsley software based.

Doesn't require a real target s/m

Very primitive

Lack of real time behavior

ADVANTAGES OF SIMULATION BASED DEBUGGING

1. No need for original target s/m

   IDE'S s/w support simulates the CPU of the target board. the real h/w is not required, firmware development can start well in advance immediately after the device interface and memory maps are finalized

2. Simulate I/O peripherals

   Simulator provides the option to simulate various I/O peripherals. Listing simulators I/O supports we can edit the value for I/O registers and can be used as the I/O value in the firmware execution. Hence it eliminates the need for connecting I/O devices for debugging the firmware.

3. Simulates abnormal condition: It really helps the developer in simulating the abnormal operational environment for firmware and helps the firmware developer to study the behaviour of the firmware under abnormal input condition.

Limitations of simulators: simulation based firmware debugging tech is very helpful in embedded application but they possesses some limitation and we cannot fully rely upon this

a)deviation from real behaviour.

b)lack of real timeliness.

in distributed embedded s/m , several processing elements(PE's) / either microprocessor or ASICS) are connected by an/to that allows them to communicate.

There are several reasons to build network- based embedded system

- When the processing tasks are physically distributed, it may be necessary to put some of the computing power near were the event occur
- Data reduction has another important reason for distributed processing. It may be possible to perform some initial signal processing on captured data to reduce its volume.
- Modularity is a motivation for n/w based design
- A distributed system can also be easier to debug
- Distributed system design is example of h/w s/w code design
- A n/w that can be used to build distributed embedded s/m

DISTRIBUTED EMBEDDED ARCHITECTURE

A distributed embedded system can be organized in many different ways but its processing statements are the basic units and networks.

An IDE provides a coherent, easy to use network environment for building applications.

IDE presents various ways of ways viewing and working with all the components that comprise the system

In terms of the task you can perform

- Organise your resources(projects folders and files).
- Edit resources
- Collaborate on projects with a team
- Compile run and debugging program
- Build OS and flash for your embedded system
- Analyse and find-tune your systems performance

Integrated Development Environments

Combines s/w editors, compilers, debuggers ,and programmers into a single IDE.

Increases productivity when doing embedded development but adds complexity and increases the learning curve of the IDE.

The care of the IDE is the integration of the s/w editor, compiler and debugger which allow a program to be rapidly debugged and monitored as the code is executed an the embedded hardware.

HEX file(.hex)

- Hex file is the binary executable file created from the source code.
- The absolute object file created by the linker/loader is converted into processor understandable binary code.
- The utility used for converting an object file to a hex file is known as object to hex file convertor
- Hex files embed the m/c code in a particular format
- The format of the hex file varies across the family of the processors /controllers
- Intel hex and Motorola hex are the two commonly used hex files formats in embedded application

EMULATORS AND DEBUGGERS

What is debugging?

Debugging in embedded application is the process of diagnosing the firmware execution, monitoring the target processor's registers and memory while the firmware is running and checking the signals from various buses of the embedded h/w

Debugging process

- Hardwaredebugging.
- Firmware debugging.

H/w debugging:deals with the monitoring of various bus signals and checking the status lines of the target h/w.

Firmware debugging: deals with examining the firmware execution, execution flow, changes to the various CPU registers and status registers on execution of the firmware to ensure that the firmware is running as per the design.

Why is Debugging required?

Firmware debugging is performed to configure out the bug or the error in the firmware which creates the accepted behaviour.

EMULATORS AND DEBUGGERS

Firmware debugging is performed to figure out the bug or error in the firmware which creates the unexpected behaviour.

Firmware is analogous to the human body in the sense it is widespread and/or modular. Any abnormal by in any area of the body may lead to sickness.

During the early days of embedded system development there were no debug tools available and the only way was "burn the code in an EEPROM and pray for its proper functioning" if the product crashes the developer is unlucky and he needs to sit back rework on the firmware till the product function in the expected ways.