Francis Xavier Engineering College

Tirunelveli – 627003

A Course Material

on

19CS7707 REAL TIME EMBEDDED SYSTEMS

By

S.MALAIARASAN

Assistant Professor

Department OF COMPUTER SCIENCE & ENGINEERING

Subject Code: **19CS7707**

Subject Name: **REAL TIME EMBEDDED SYSTEMS**

Year/Sem: IV/VII

Signature of the Author

Name: Mr.S.Malaiarasan

Designation: Assistant Professor

EC6703 EMBEDDED AND REAL TIME SYSTEMS L T P C 3 0 0 3

OBJECTIVES:

The student should be made to:

- Learn the architecture and programming of ARM processor.
- Be familiar with the embedded computing platform design and analysis.
- Be exposed to the basic concepts of real time Operating system.
- Learn the system design techniques and networks for embedded systems

UNIT II NTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS 9

Complex systems and micro processors– Embedded system design process –Design example: Model train controller- Instruction sets preliminaries - ARM Processor – CPU: programming input and output supervisor mode, exceptions and traps – Co-processors- Memory system mechanisms – CPU performance- CPU power consumption.

UNIT II EMBEDDED COMPUTING PLATFORM DESIGN 9

The CPU Bus-Memory devices and systems–Designing with computing platforms – consumer electronics architecture – platform-level performance analysis - Components for embedded programs Models of programs- Assembly, linking and loading – compilation techniques- Program level performance analysis – Software performance optimization – Program level energy and power analysis and optimization – Analysis and optimization of program size- Program validation and testing.

UNIT III PROCESSES AND OPERATING SYSTEMS 9

Introduction – Multiple tasks and multiple processes – Multirate systems- Preemptive real-time operating systems- Priority based scheduling- Interposes communication mechanisms – Evaluating operating system performance- power optimization strategies for processes – Example Real time operating systems- POSIX- Windows CE.

UNIT IV SYSTEM DESIGN TECHNIQUES AND NETWORKS 9

Design methodologies- Design flows - Requirement Analysis – Specifications- System analysis and architecture design – Quality Assurance techniques- Distributed embedded systems – MPSoCs and shared memory multiprocessors.

UNIT V CASE STUDY **9**

Data compressor - Alarm Clock - Audio player - Software modem-Digital still camera - Telephone answering machine-Engine control unit – Video accelerator.

TOTAL: 45 PERIODS

OUTCOMES:

Upon completion of the course, students will be able to:

- Describe the architecture and programming of ARM processor.
- Outline the concepts of embedded systems
- Explain the basic concepts of real time Operating system design.
- Use the system design techniques to develop software for embedded systems
- Differentiate between the general purpose operating system and the real time operating system
- Model real-time applications using embedded-system concepts

TEXT BOOK:

1. Marilyn Wolf, “Computers as Components - Principles of Embedded Computing System Design”, Third Edition “Morgan Kaufmann Publisher (An imprint from Elsevier), 2012.

REFERENCES:

1. Jonathan W.Valvano, “Embedded Microcomputer Systems Real Time Interfacing”, Third Edition Cengage Learning, 2012.

2. David. E. Simon, “An Embedded Software Primer”, 1st Edition, Fifth Impression, Addison-Wesley Professional, 2007.

3. Raymond J.A. Buhr, Donald L.Bailey, “An Introduction to Real-Time Systems-From Design to Networking with C/C++”, Prentice Hall, 1999.

4. C.M. Krishna, Kang G. Shin, “Real-Time Systems”, International Editions, Mc Graw Hill 1997

5. K.V.K.K.Prasad, “Embedded Real-Time Systems: Concepts, Design & Programming”, Dream Tech Press, 2005.

6. Sriram V Iyer, Pankaj Gupta, “Embedded Real Time Systems Programming”, Tata Mc Graw Hill, 2004.

CONTENTS

S.No	Particulars	Page
1	Unit – I	06
2	Unit – II	41
3	Unit – III	75
4	Unit – IV	93
5	Unit – V	114

Unit-I

Introduction to embedded computing and arm processors

Part – A

1. Differentiate top-down and bottom-up design. [CO1-L3-April2014]

Top-Down:

Top down design proceeds from the abstract entity to get to the concrete design.

It is most often used in designing brand new systems.

Bottom-Up:

Bottom-up design proceeds from the concrete design to get to the abstract entity.

It is sometimes used when one is reverse engineering a design, (i.e) when one is trying to

figure out what somebody else designed in an existing design.

2. List the functions of ARM processor in supervisor mode. [CO1-L2-April 2014]

The various functions of ARM processor in supervisor modes are:

- ✓ Exception
- ✓ Prioritization
- ✓ Vectoring
- ✓ Traps.

3. Enumerate various issues in real time computing. [CO1-L3-Nov/Dec 2013]

The various issues in Real Time computing is:

- ✓ Real -time Response
- ✓ Recovering from failures
- ✓ Working with distributed architecture
- ✓ Asynchronous communication
- ✓ Race condition and timing.

4. Write short notes on ARM Processor. [CO1-L1-Nov/Dec 2013]

- ✓ ARM – Advanced RISC Machine. (It is an 32-bit Microprocessor)
- ✓ ARM is actually a family of RISC architectures thathave been developed over many years.
- ✓ The ARM is a 32-bit Reduced Instruction Set Computer (RISC) instruction set architecture developed by Arm holdings.
- ✓ ARM processor is mad suitable for Low power application.

5. What are the Instruction set features useful for embedded programming? [CO1-L1-May/June 2013]

Instruction Sets can have a variety of characteristics/features including:

- ✓ Fixed versus variable length
- ✓ Addressing modes
- ✓ Number of operands
- ✓ Types of operations supported.

6. What are the parameters used to evaluate the CPU performance? [CO1-L1-May/June 2013]

- ✓ Pipelining
- ✓ Caching.

7. What is the function of exception? [CO1-L1-Nov/Dec 2012]

- ✓ The main function of exception is to detect the error internally.
- ✓ It requires both prioritization and vectoring.

8. How is ARM processor different from other processors? [CO1-H2-Nov/Dec 2012]

ARM is a RISC (Reduced Instruction Set Computing) architecture while other processor
being a CISC (Complex Instruction Set Computing) one.

In the ARM processor, arithmetic and logical operations cannot be perform directly on memory locations, while other processors allow such operations to directly reference main memory.

9. When is application specific system processor (ASSPs) used in a embedded systems? [CO1-L1-May/June 2012]

ASSP is a processing unit for specific task and for specific application. In embedded system for example image compression and that is integrated through the buses with the main processor in an embedded system.

10. What are the various in embedded system designs modelling refining (or) partitioning? [CO1-L1-May/June 2012]

- ✓ Structural modelling
- ✓ Behaviour modelling
- ✓ State machine modelling
- ✓ Process algebra modelling
- ✓ Logic based modelling
- ✓ Petri-nets modelling.

11. What is embedded system? [CO1-L1]

An Embedded system is one that has computer hardware with software embedded in it as one of its most important component.

12. What are the components of embedded system? [CO1-L1]

Hardware, Main Application Software, RTOS

13. What are the requirements of embedded system? [CO1-L1]

- ✓ Reliability
- ✓ Low Power Consumption
- ✓ Cost Effectiveness
- ✓ Efficient Use of Processing Power

14. What is microprocessor? [CO1-L1]

A microprocessor is a single VLSI chip that has a CPU and may also have some other units for example floating point processing arithmetic unit pipelining and super scaling units for faster processing of instruction.

15. Give the characteristics of embedded system? [CO1-L1]

- ✓ Perform a single specific task.
- ✓ Time bounded.
- ✓ Contains atleast one programmable unit. (micro-controller)
- ✓ Runs a single program.
- ✓ Minimum cost

16. What are the challenges of embedded system? [CO1-L1]

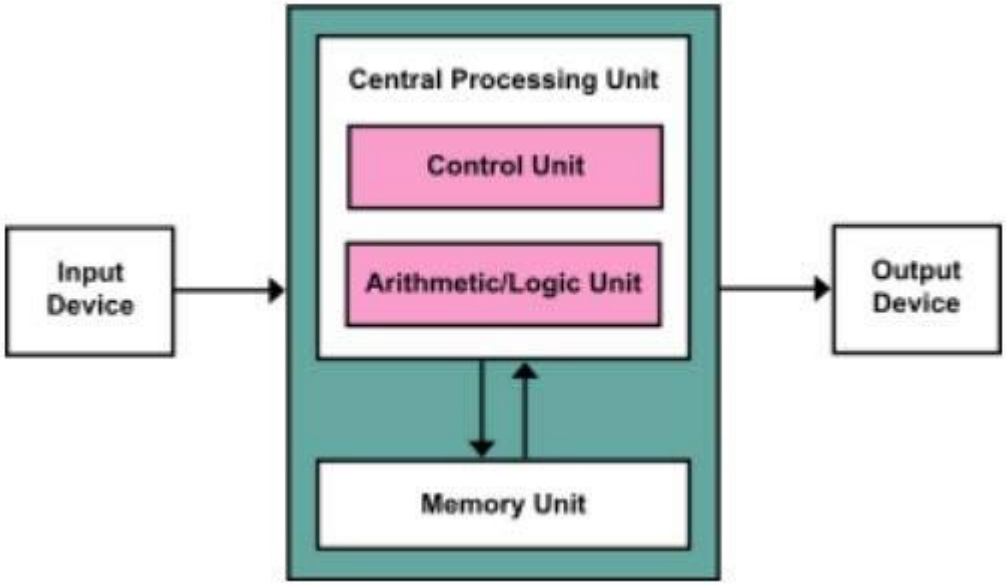
- ✓ Hardware needed
- ✓ Meeting the deadlines
- ✓ Minimizing the power consumption
- ✓ Design for upgradeability

17. Give the steps in embedded system design? [CO1-L1]

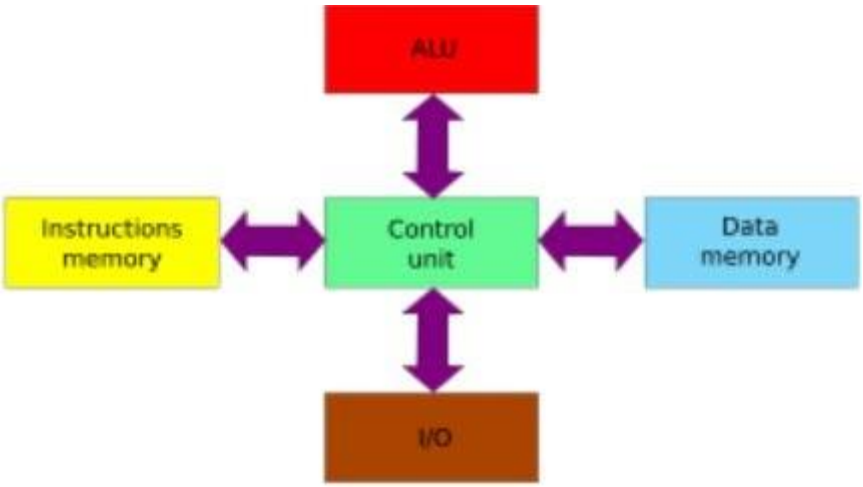
- ✓ Requirements
- ✓ Specifications
- ✓ Architecture
- ✓ Components
- ✓ System Integration

18. Draw Von-Neumann and Harvard architecture? [CO1-L1]

Von-Neumann Architecture :



Harvard architecture:



19. What are the functions of memory? [CO1-L1]

The basic function of computer memory is essentially to store data. Depending on the type of data it stores and the role it plays in computer operation, however, memory performs several different functions. Although all of these functions involve data storage, RAM, ROM, flash memory and hard drives each perform a different and necessary function to keep a computer and its peripherals working.

20. Define ARM Processor? [CO1-L1]

An ARM processor is any of several 32-bit RISC (reduced instruction set computer) microprocessor s developed by Advanced RISC Machines, Ltd. The ARM architecture was originally conceived by Acorn Computers Ltd. in the 1980s. Since then, it has evolved into a family of microprocessors extensively used in consumer

electronic devices such as mobile phone s, multimedia players, pocket calculators and PDA s (personal digital assistants).

21. What are data types supported by ARM? [CO1-L1]

Standard ARM word is 32 bit long Word is splitted into 4, 8 bit bytes

22. What is the use of requirements form?

To collect the following information

- ✓ Name
- ✓ Purpose
- ✓ Inputs & outputs
- ✓ Functions
- ✓ Performance
- ✓ Manufacturing cost
- ✓ Power
- ✓ Physical size and weight

23. Define RISC? [CO1-L1]

RISC, or Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

Part-B

1. Explain about complex systems and microprocessors. (10) [CO1-L1]

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.

This means that embedded computing system design is a useful skill for many types of product design. Automobiles, cell phones, and even household appliances make extensive use of microprocessors. Designers in many fields must be able to identify where microprocessors can be used, design a hardware platform with I/O devices that can support the required tasks, and implement software that performs the required processing.

Computer engineering, like mechanical design or thermodynamics, is a fundamental discipline that can be applied in many different domains. But of course, embedded computing system design does not stand alone.

Many of the challenges encountered in the design of an embedded computing system are not computer engineering—for example, they may be mechanical or analog electrical problems. In this book we are primarily interested in the embedded computer itself, so we will concentrate on the hardware and software that enable the desired functions in the final product.

Embedding Computers

Computers have been embedded into applications since the earliest days of computing. One example is the Whirlwind, a computer designed at MIT in the late 1940s and early 1950s. Whirlwind was also the first computer designed to support **real-time** operation and was originally conceived as a mechanism for controlling an aircraft simulator.

Even though it was extremely large physically compared to today's computers (e.g., it contained over 4,000 vacuum tubes), its complete design from components to system was attuned to the needs of real-time embedded computing.

The utility of computers in replacing mechanical or human controllers was evident from the very beginning of the computer era—for example, computers were proposed to control chemical processes in the late 1940s.

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.

However, 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.

Automobile designers started making use of the microprocessor soon after single-chip CPUs became available.

The most important and sophisticated use of microprocessors in automobiles was to control the engine: determining when spark plugs fire, controlling the fuel/air mixture, and so on. There was a trend toward electronics in automobiles in general—electronic devices could be used to replace the mechanical distributor.

But the big push toward microprocessor-based engine control came from two nearly simultaneous developments: The oil shock of the 1970s caused consumers to place

much higher value on fuel economy, and fears of pollution resulted in laws restricting automobile engine emissions.

The combination of low fuel consumption and low emissions is very difficult to achieve; to meet these goals without compromising engine performance, automobile manufacturers turned to sophisticated control algorithms that could be implemented only with microprocessors.

Microprocessors come in many different levels of sophistication; they are usually classified by their word size. An 8-bit **microcontroller** is designed for low-cost applications and includes on-board memory and I/O devices; a 16-bit microcontroller is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory; and a 32-bit **RISC** microprocessor offers very high performance for computation-intensive applications.

Given the wide variety of microprocessor types available, it should be no surprise that microprocessors are used in many ways. There are many household uses of microprocessors. The typical microwave oven has at least one microprocessor to control oven operation.

Many houses have advanced thermostat systems, which change the temperature level at various times during the day. The modern camera is a prime example of the powerful features that can be added under microprocessor control.

Digital television makes extensive use of embedded processors. In some cases, specialized CPUs are designed to execute important algorithms—an example is the CPU designed for audio
processing in the SGS Thomson chip set for DirecTV [Lie98]. This processor is designed to efficiently implement programs for digital audio decoding.

A programmable CPU was used rather than a hardwired unit for two reasons: First, it made the system easier to design and debug; and second, it allowed the possibility of upgrades and using the CPU for other purposes.

A high-end automobile may have 100 microprocessors, but even inexpensive cars today use 40 microprocessors. Some of these microprocessors do very simple things such as detect whether seat belts are in use. Others control critical functions such as the ignition and braking systems.

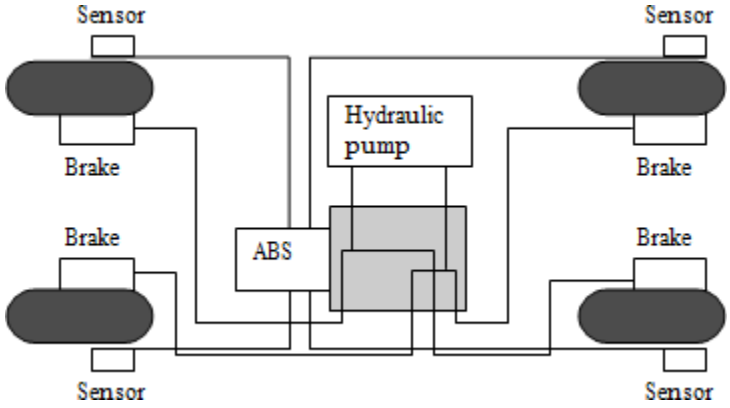
2. Describe the architecture and functions of BMW 850i brake and stability control system. (8) [CO1-L1]

The BMW 850i was introduced with a sophisticated system for controlling the wheels of the car. An antilock brake system (ABS) reduces skidding by pumping the brakes.

An automatic stability control (ASC +T) system intervenes with the engine during manoeuvring to improve the car’s stability. These systems actively control critical systems of the car; as control systems, they require inputs from and output to the automobile.

Let's first look at the ABS. The purpose of an ABS is to temporarily release the brake on a wheel when it rotates too slowly—when a wheel stops turning, the car starts skidding and becomes hard to control. It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves as seen in the following diagram. This hookup allows the ABS system to modulate the brakes in order to keep the wheels from locking.

The ABS system uses sensors on each wheel to measure the speed of the wheel. The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent the wheels from skidding.



The ASC + T system's job is to control the engine power and the brake to improve the car's stability during maneuvers

The ASC+T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting.

The ASC + T can be turned off by the driver, which can be important when operating with tire snow chains.

The ABS and ASC+ T must clearly communicate because the ASC + T interacts with the brake system. Since the ABS was introduced several years earlier than the ASC + T, it was important to be able to interface ASC + T to the existing ABS module, as well as to other existing electronic modules.

The engine and control management units include the electronically controlled throttle, digital engine management, and electronic transmission control. The ASC + T control unit has two microprocessors on two printed circuit boards, one of which concentrates on logic-relevant components and the other on performance-specific components.

3. List the Characteristics of Embedded Computing Applications. (6) [CO1-L1]

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.

On the one hand, embedded computing systems have to provide sophisticated functionality:

- 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 optimize 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.

To make things more difficult, embedded computing operations must often be performed to meet deadlines:

- 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.

Costs of various sorts are also very important:

- 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 consumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

4. Explain the various Steps required designing an embedded system. (12) [CO1-L1]

This section provides an overview of the embedded system design process aimed at two objectives. First, it will give us an introduction to the various steps in embedded system design before we delve into them in more detail. Second, it will allow us to consider the design **methodology** itself. 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 performing 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 semi automating) the steps one at a time 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.

Figure summarizes the major steps in the embedded system design process. In this top–down view, we start with the system **requirements**. In the next step, **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.

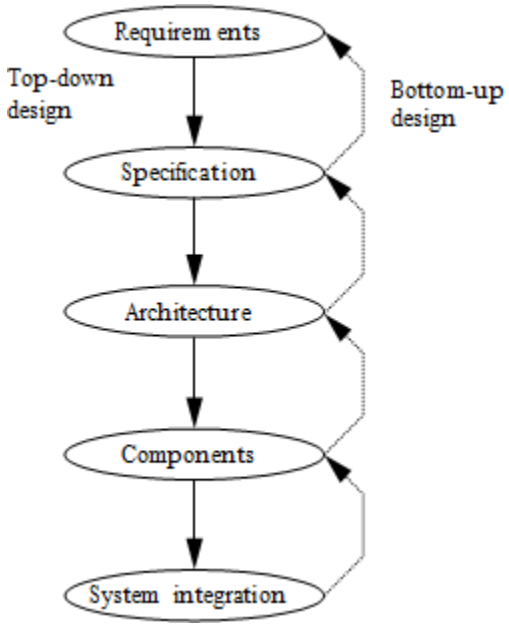


Fig Major levels of abstraction in the design process.

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.
- 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.

Separating out requirements analysis and specification is often necessary because of the large gap between what the customers can describe about the system they want and what the architects need to design the system.

Consumers of embedded systems are usually not themselves embedded system designers or even product designers. Their understanding of the system is based on how they envision users' interactions with the system. They may have unrealistic expectations as to what can be done within their budgets; and they may also express their desires in a language very different from system architects' jargon.

Capturing a consistent set of requirements from the customer and then massaging those requirements into a more formal specification is a structured way to manage the process of translating from the consumer's language to the designer's.

Requirements may be **functional** or **non functional**. We must of course capture the basic functions of the embedded system, but functional description is often not sufficient. Typical non functional requirements include:

- 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, performance 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 engineering (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 mockup 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, non functional models of devices can also give customers a better idea of characteristics such as size and weight.

Name

- Purpose
- Inputs
- Outputs
- Functions
- Performance
- Manufacturing cost
- Power
- Physical size and weight

Sample requirements form.

Requirements analysis for big systems can be complex and time consuming. However, capturing a relatively small amount of information in a clear, simple format is a good start toward understanding system requirements.

To introduce the discipline of requirements analysis as part of system design, we will use a simple requirements methodology.

Figure 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 you don't understand it well enough.

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

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

— Data characteristics: Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?
— Types of I/O devices: Buttons? Analog/digital converters? Video displays?

- Functions: This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?
- Performance: Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time frame. It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.
- Manufacturing cost: This includes primarily the cost of the hardware 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.

5. Design an embedded system which controls the track of handling eight trains (Model train control)(12) [CO1-L1]

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 1.2.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.

We start by analyzing the requirements for the train control system.We will base our system on a real standard developed for model trains.We then develop two specifications: a simple, high-level specification and then a more detailed specification.

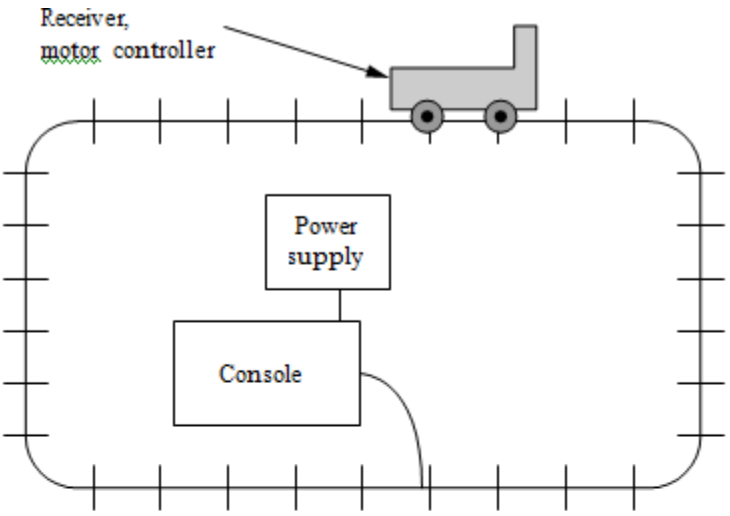
Requirements

- Before we can create a system specification, we have to understand the requirements.
- 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 responsiveness 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.

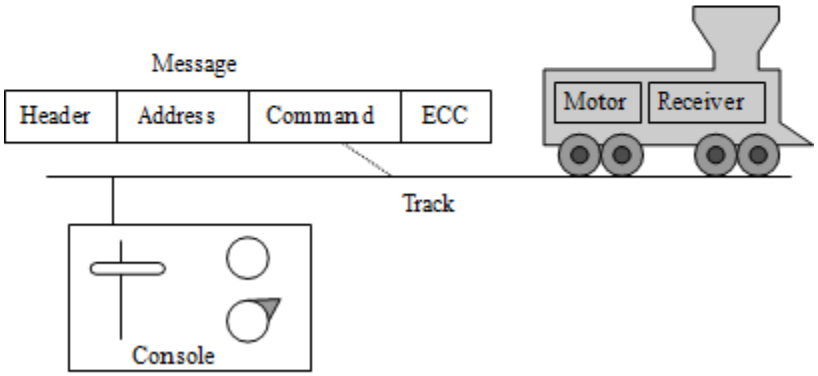
We can put the requirements into chart format:

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
Functions	Set engine speed based upon inertia settings; respond to emergency stop
Performance	
Manufacturing cost	Can update train speed at least 10 times per second
Power	$50
Physical size and weight	10W (plugs into wall)
	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.



System setup



Signaling the train

DCC

The **Digital Command Control (DCC)** was created by the National Model Railroad Association to support interoperable digitally-controlled model trains.

Hobbyists 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 language 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 interoperability. Over standardization, 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.

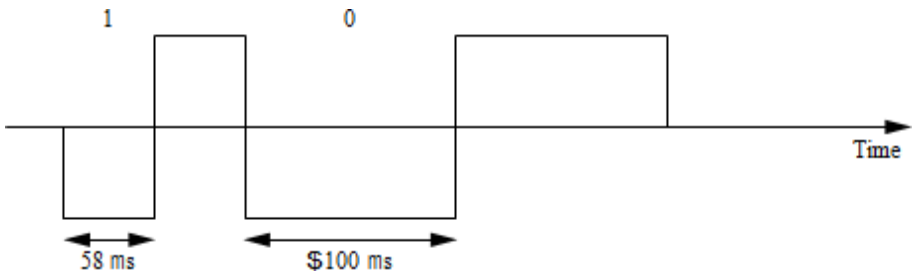
The data signal swings between two voltages around the power supply voltage. As shown in Figure , bits are encoded in the time between transitions, not by voltage levels. A 0 is at least 100 ms while a 1 is nominally 58ms.

The durations 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. We can write the basic packet format as a regular expression:



Bit encoding in DCC

PSA (sD) + E

In this regular expression:

- 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 document.

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 instruction; 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 data byte 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.

6. Brief about Instruction sets preliminaries in embedded system environment.(6) [CO1-L1]

Computer Architecture Taxonomy

Before we delve into the details of microprocessor instruction sets, it is helpful to develop some basic terminology. We do so by reviewing taxonomy of the basic ways we can organize a computer.

A block diagram for one type of computer is shown in Figure The computing system consists of a **central processing unit (CPU)** and a **memory**.

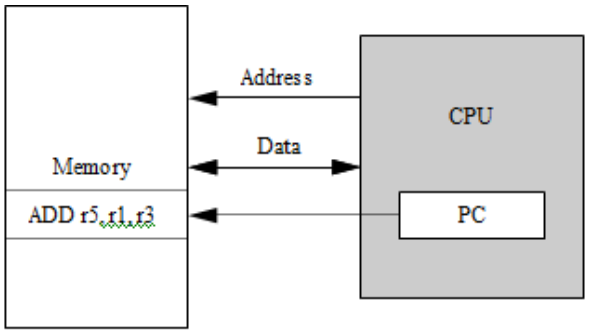
The memory holds both data and instructions, and can be read or written when given an address. A computer whose memory holds both data and instructions is known as a **von Neumann** machine.

The CPU has several internal **registers** that store values used internally. One of those registers is the **program counter (PC)**, which holds the address in memory of an instruction. The CPU fetches the instruction from memory, decodes the instruction, and executes it.

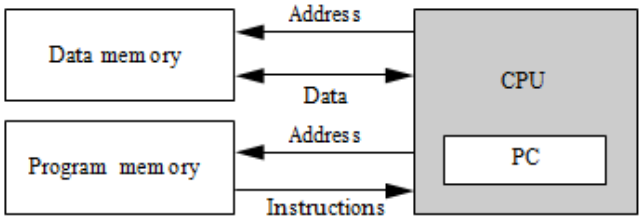
The program counter does not directly determine what the machine does next, but only indirectly by pointing to an instruction in memory. By changing only the instructions, we can change what the CPU does. It is this separation of the instruction memory from the CPU that distinguishes a stored-program computer from a general finite-state machine.

An alternative to the von Neumann style of organizing computers is the **Harvard architecture**, which is nearly as old as the von Neumann architecture. As shown in Figure, a Harvard machine has separate memories for data and program.

The program counter points to program memory, not data memory. As a result, it is harder to write self-modifying programs (programs that write data values, and then use those values as instructions) on Harvard machines.



A von Neumann architecture computer.



A Harvard architecture.

Harvard architectures are widely used today for one very simple reason—the separation of program and data memories provides higher performance for digital signal processing.

Processing signals in real-time places great strains on the data access system in two ways: First, large amounts of data flow through the CPU; and second, that data must be

processed at precise intervals, not just when the CPU gets around to it. Data sets that arrive continuously and periodically are called **streaming data**.

Having two memories with separate ports provides higher memory bandwidth; not making data and memory compete for the same port also makes it easier to move the data at the proper times. DSPs constitute a large fraction of all microprocessors sold today, and most of them are Harvard architectures.

A single example shows the importance of DSP: Most of the telephone calls in the world go through at least two DSPs, one at each end of the phone call.

Another axis along which we can organize computer architectures relates to their instructions and how they are executed. Many early computer architectures were what is known today as **complex instruction set computers (CISC)**. These machines provided a variety of instructions that may perform very 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)**.These computers tended to provide somewhat 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.

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

- There may be several different implementations of architecture. In fact, the architecture definition serves to define those characteristics that must be true of all implementations and what may vary from implementation to implementation.

- Different CPUs may offer different clock speeds, different cache configurations, changes to the bus or interrupt lines, and many other changes that can make one model of CPU more attractive than another for any given application.

7. Write short note on Assembly language. (6) [CO1-L1]

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

- One instruction appears per line.
- **Labels**, which give names to memory locations, start in the first column.
- Instructions must start in the second column or after to distinguish them from labels.
- Comments run from some designated comment character (; in the case of ARM) to the end of the line.

Assembly language follows this relatively structured form to make it easy for the **assembler** to parse the program and to consider most aspects of the program line by line. (It should be remembered that early assemblers were written in assembly language to fit in a very small amount of memory.

Those early restrictions have carried into modern assembly languages by tradition.) Figure 2.4 shows the format of an ARM data processing instruction such as an ADD.

ADD r0, r3, #5

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

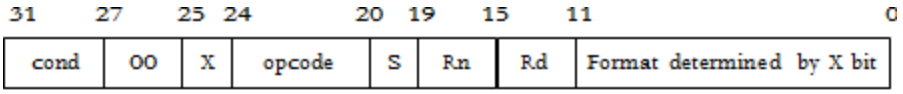
```
label1   ADR r4,c
         LDR r0,[r4]        ; a comment
         ADR r4,d
         LDR r1,[r4]
         SUB r0,r0,r1       ; another comment
```

An example of ARM assembly language.

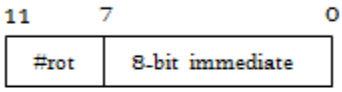
Assemblers must also provide some **pseudo-ops** to help programmers create complete assembly language programs.

An example of a pseudo-op is one that allows data values to be loaded into memory locations. These allow constants, for example, to be set into memory.

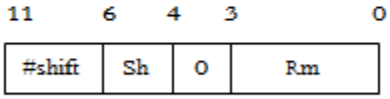
An example of a memory allocation pseudo-op for ARM is shown in Figure .TheARM % pseudo-op allocates a block of memory of the size specified by the operand and initializes those locations to zero.

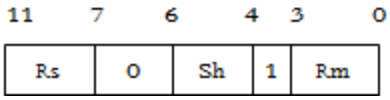


X51 (represents operand 2):



X50 format:





8. What is ARM? Also elaborate the memory and instructions of ARM. (10) [CO1-L2]

. ARM is actually a family of RISC architectures that have been developed over many years.

ARM does not manufacture its own VLSI devices; rather, it licenses its architecture to companies who either manufacture the CPU itself or integrate the ARM processor into a larger system.

The textual description of instructions, as opposed to their binary representation, is called an assembly language.

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 in the first column. Here is an example:

LDR r0, [r8]; a comment
label ADD 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 Harvard architecture.

However, this difference is invisible to the assembly language programmer, except for possible performance differences.

The ARM architecture supports two basic types of data: The standard ARM word is 32 bits long.

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

ARM7 allows addresses up to 32 bits long. An address refers to a byte, not a word. Therefore, the word 0 in the ARM address space is at location 0, the word 1 is at 4, the word 2 is at 8,and so on. (As a result, the PC is incremented by 4 in the absence of a branch.)

The ARM processor can be configured at power-up to address the bytes in a word in either **little-endian** mode (with the lowest-order byte residing in the low-order bits of the word) or **big-endian** mode (the lowest-order byte stored in the highest bits of the word), as illustrated in Figure 1.14 [Coh81]. General purpose computers have sophisticated instruction sets.

Some of this sophistication is required simply to provide the functionality of a general computer, while other aspects of instruction sets may be provided to increase performance, reduce code size, or otherwise improve program characteristics.

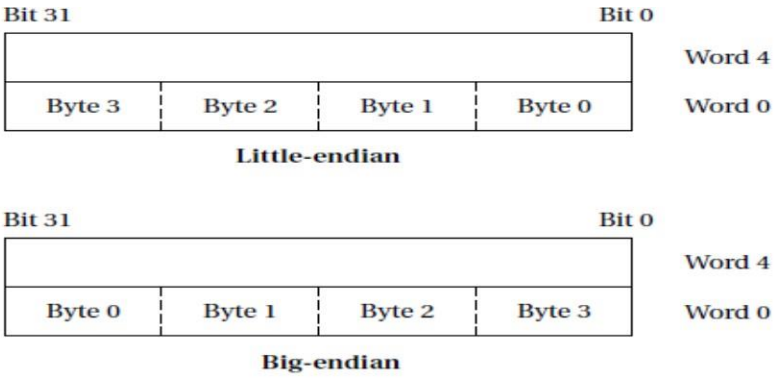


Fig 1.14

Byte organizations within an ARM word.

Data Operations:

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

Figure 1.15 shows a sample fragment of C code with data declarations and several assignment statements. The variables a, b, c, x, y, and z all become data locations in memory. In most cases data are kept relatively separate from instructions in the program's memory image.

In the ARM processor, arithmetic and logical operations cannot be performed directly on memory locations. While some processors allow such operations to directly reference main memory, ARM is a **load-store architecture**—data operands must first be loaded into the CPU and then stored back to main memory to save the results. Figure 2.8 shows the registers in the basic ARM programming model. ARM has 16 general-purpose registers, r0 through r15. Except for r15, they are identical—any operation that can be done on one of them can be done on the other one also.

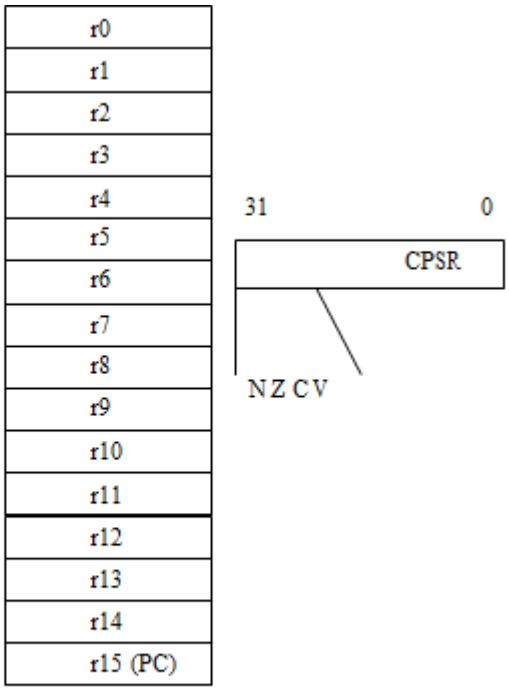
The r15 register has the same capabilities as the other registers, but it is also used as the program counter. The program counter should of course not be overwritten for use in data operations. However, giving the PC the properties of a general-purpose register allows the program counter value to be used as an operand in computations, which can make certain programming tasks easier. The other important basic register in the programming model is the **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.

```
int a, b, c, x, y, z;
x = (a+b)-c;
y=a*(b+c);
z=(a << 2) | (b & 15);
```

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



The basic ARM programming model.

The basic form of a data instruction is simple:

ADD r0,r1,r2

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

ADD r0,r1,#2
sets r0 to r1+2.

The major data operations are summarized in Figure 1.17. 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: No operand may be an immediate, and the two source operands must be different registers.

The MLA instruction performs a multiply-accumulate operation, particularly useful in matrix operations and signal processing. The instruction
MLA r0, r1, r2, r3
Sets r0 to the value r1*r2+r3.

The shift operations are not separate instructions rather; shifts can be applied to arithmetic and logical instructions. The shift modifier is always applied to the second source operand.

A left shift moves bits up toward the most-significant bits, while a right shift moves bits down to the least-significant bit in the word.

The LSL and LSR modifiers perform left and right logical shifts, filling the least-significant bits of the operand with zeroes. The arithmetic shift left is equivalent to an LSL, but the ASR copies the sign bit—if the sign is 0, a 0 is copied, while if the sign is 1, a 1 is copied.

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

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

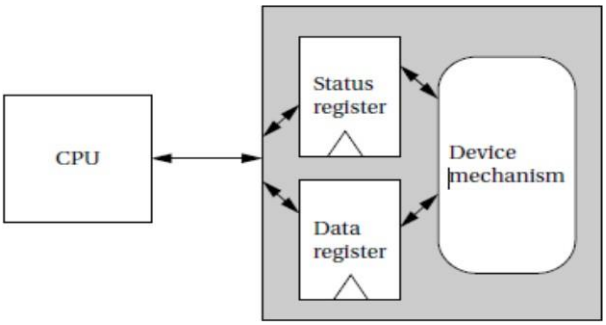
Logical

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

9. Write short notes on input and output device interfacing with CPU. (6) [CO1-L1]

The basic techniques for I/O programming can be understood relatively independent of the instruction set. In this section, we cover the basics of I/O programming and place them in the contexts of both the ARM and C55x.

We begin by discussing the basic characteristics of I/O devices so that we can understand the requirements they place on programs that communicate with them.



Input and 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. But the digital logic in the device that is most closely connected to the CPU very strongly resembles the logic you would expect in any computer system.

Figure shows the structure of a typical I/O device and its relationship to the CPU.The interface between the CPU and the device's internals (e.g.,the rotating disk and read/write electronics in a disk drive) is a set of registers. The CPU talks to the device by reading and writing the registers.

Devices typically have several registers:

Data registers hold values that are treated as data by the device, such as the data read or written by a disk.

Status registers provide information about the device's operation, such as whether the current transaction has completed.

Some registers may be read-only, such as a status register that indicates when the device is done, while others may be readable or writable.

Input and 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.

Busy-Wait I/O:

The most basic way to use devices in a program is busy-wait I/O. Devices are typically slower than the CPU and may require many cycles to complete 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. (If we try to start writing the second character before the device has finished with the first one, for example,

the device will probably never print the first character.) Asking an I/O device whether it is finished by reading its status register is often called polling.

10. Describe about supervisor mode, exceptions, and traps. (6) [CO1-H1]

These are mechanisms to handle internal conditions, and they are very similar to interrupts in form. We begin with a discussion of supervisor mode, which some processors use to handle exceptional events and protect executing programs from each other.

Supervisor Mode:

As will become clearer in later chapters, complex systems are often implemented as several programs that communicate with each other. These programs may run under the command of an operating system. It may be desirable to provide hardware checks to ensure that the programs do not interfere with each other—for example, by erroneously writing into a segment of memory used by another program. Software debugging is important but can leave some problems in a running system; hardware checks ensure an additional level of safety.

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. 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.

Not all CPUs have supervisor modes. Many DSPs, including the C55x, do not provide supervisor modes. The ARM, however, does have such a mode. The ARM instruction that puts the CPU in supervisor mode is called SWI:

SWI CODE_1

It can, of course, be executed conditionally, as with any ARM instruction. SWI causes the CPU to go into supervisor mode and sets the PC to 0x08.The argument to SWI is a 24-bit immediate value that is passed on to the supervisor mode code; it allows the program to request various services from the supervisor mode.

In supervisor mode, the bottom 5 bits of the CPSR are all set to 1 to indicate that the CPU is in supervisor mode. The old value of the CPSR just before the SWI is stored in a register called the **saved program status register (SPSR)**. There are in fact several SPSRs for different modes; the supervisor mode SPSR is referred to as SPSR_svc.

To return from supervisor mode, the supervisor restores the PC from register r14 and restores the CPSR from the SPSR_svc.

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.

Just as interrupts can be seen as an extension of the subroutine mechanism, exceptions are generally implemented as a variation of an interrupt. Since both deal with changes in the flow of control of a program, it makes sense to use similar mechanisms. However, exceptions are generated internally.

Exceptions in general require both prioritization and vectoring. Exceptions must be prioritized because a single operation may generate more than one exception for example, an illegal operand and an illegal memory access.

The priority of exceptions is usually fixed by the CPU architecture. Vectoring provides a way for the user to specify the handler for the exception condition.

The vector number for an exception is usually predefined by the architecture; it is used to index into a table of exception handlers.

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.

11. What is co-processors? How it s used in embedded application? (4) [CO1-L1]

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, floating-point 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 instruction, the CPU must activate the co-processor and pass it the relevant instruction. Co-processor instructions can 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.
A CPU may, of course, receive co-processor instructions even when there is no coprocessor attached. Most architectures use illegal instruction traps to handle these situations. The trap handler can detect the co-processor instruction and, for example, execute it in software on the main CPU. Emulating co-processor instructions in software is slower but provides compatibility.
The ARM architecture provides support for up to 16 co-processors. Co-processors are able to perform load and store operations on their own registers. They can also move data between the co-processor registers and main ARM registers.
An example ARM co-processor is the floating-point unit. The unit occupies two co-processor units in the ARM architecture, numbered 1 and 2, but it appears as a single unit to the programmer. It provides eight 80-bit floating-point data registers, floating-point status registers, and an optional floating-point status register.

12. Explain the memory system mechanisms to increase the performance in an embedded system. (12) [CO1-H1]

Modern microprocessors do more than just read and write a monolithic memory. Architectural features improve both the speed and capacity of memory systems.

Microprocessor clock rates are increasing at a faster rate than memory speeds, such that memories are falling further and further behind microprocessors every day. As a result, computer architects resort to **caches** to increase the average performance of the memory system.

Although memory capacity is increasing steadily, program sizes are increasing as well, and designers may not be willing to pay for all the memory demanded by an application. **Modern microprocessor units (MMUs)** perform address translations that provide a larger virtual memory space in a small physical memory. In this section, we review both caches and MMUs.

Caches:

Caches are widely used to speed up memory system performance. Many microprocessor architectures include caches as part of their definition.

The cache speeds up average memory access time when properly used. It increases the variability of memory access times accesses in the cache will be fast, while access to locations not cached will be slow. This variability in performance makes it especially important to understand how caches work so that we can better understand how to predict cache performance and factor variabilities into system design.

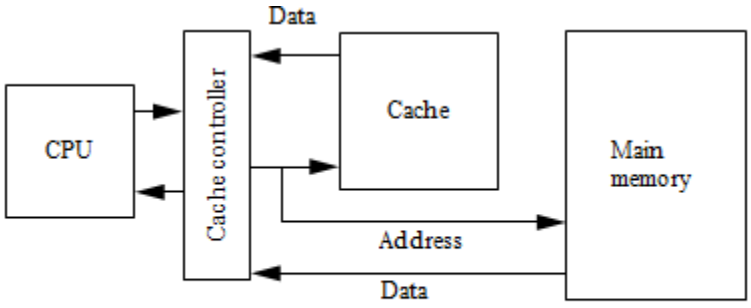
A cache is a small, fast memory that holds copies of some of the contents of main memory. Because the cache is fast, it provides higher-speed access for the CPU; but since it is small, not all requests can be satisfied by the cache, forcing the system to wait for the slower main memory. Caching makes 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.

Figure shows how the cache support reads in the memory system. 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.



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, and

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

Even before we consider ways to implement caches, we can write some basic formulas for memory system performance. Let h be the hit rate, the probability that a given memory location is in the cache. It follows that 1_h is the miss rate, or the probability that the location is not in the cache. Then we can compute the average memory access time as

tav =htcache + (1 - h)tmain.

where tcache is the access time of the cache and tmain is the main memory access time. The memory access times are basic parameters available from the memory manufacturer.

The hit rate depends on the program being executed and the cache organization, and is typically measured using simulators. The best-case memory access time (ignoring cache controller overhead) is tcache, while the worst-case access time is tmain. Given that tmain is typically 50–60 ns for DRAM, while tcache is at most a few nanoseconds, the spread between worst-case and best-case memory delays is substantial.

where tcache is the access time of the cache and tmain is the main memory access time. The memory access times are basic parameters available from the memory manufacturer. The hit rate depends on the program being executed and the cache organization, and is typically measured using simulators.

The best-case memory access time (ignoring cache controller overhead) is tcache, while the worst-case access time is tmain. Given that tmain is typically 50–60 ns for DRAM,while tcache is at most a few nanoseconds, the spread between worst-case and best-case memory delays is substantial.

Modern CPUs may use multiple levels of cache as shown in Figure 1.20. The **first-level cache** (commonly known as **L1 cache**) is closest to the CPU, the **second-level cache (L2 cache)** feeds the first-level cache, and so on.

The second-level cache is much larger but is also slower. If h1 is the first-level hit rate and h2 is the rate at which access hit the second-level cache but not the first-level cache, then the average access time for a two-level cache system is

tav = h1tL1 + h2tL2 +(1 - h1 - h2)tmain.

As the program's working set changes, we expect locations to be removed from the cache to make way for new locations. When set-associative caches are used, we have to think about what happens when we throw out a value from the cache to make room for a new value.

We do not have this problem in direct-mapped caches because every location maps onto a unique block, but in a set-associative cache we must decide which set will have its block thrown out to make way for the new block.

One possible replacement policy is least recently used (LRU), that is, throw out the block that has been used farthest in the past. We can add relatively small amounts of hardware to the cache to keep track of the time since the last access for each block. Another policy is random replacement, which requires even less hardware to implement.

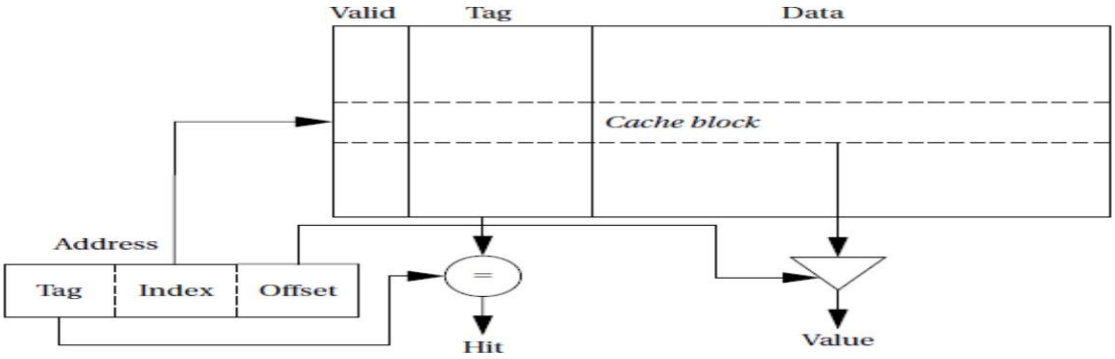
The simplest way to implement a cache is a **direct-mapped cache**, as shown in Figure 1.20. The cache consists of cache **blocks**, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid. An address is divided into three sections.

The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index. If the address tag matches the tag value in the block, that block includes the desired memory location.

If the length of the data field is longer than the minimum addressable unit, then the lowest bits of the address are used as an offset to select the required value from the data field. Given the structure of the cache, there is only one block that must be checked to see whether a location is in the cache—the index uniquely determines that block. If the access is a hit, the data value is read from the cache.

• • Write s a re s lig htly more complicated than reads because we have to update main memory as well as the cache. There are several methods by which we can do this. The simplest scheme is known as **write-through**—every write changes both the cache and the corresponding main memory location (usually through a write buffer).

This scheme ensures that the cache and main memory are consistent, but may generate some additional main memory traffic. We can reduce the number of times we write to main memory by using a **write-back** policy: If we write only when we remove a location from the cache, we eliminate the writes when a location is written several times before it is removed from the cache.



The direct-mapped cache is both fast and relatively low cost, but it does have limits in its caching power due to its simple scheme for mapping the cache onto main memory. Consider a direct-mapped cache with four blocks, in which locations 0, 1, 2, and 3 all map to different blocks. But locations 4, 8, 12…all map to the same block as location 0; locations 1, 5, 9, 13…all map to a single block; and so on. If two popular locations in a program happen to map onto the same block, we will not gain the full benefits of the cache. As seen in Section 5.6, this can create program performance problems.

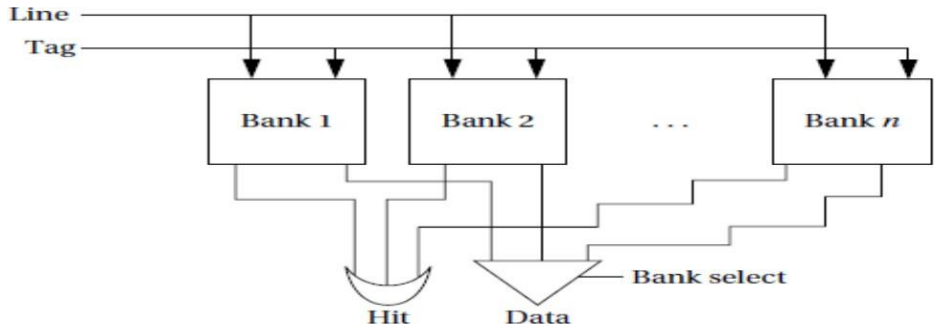
The limitations of the direct-mapped cache can be reduced by going to the **set-associative** cache structure shown in Figure 1.21.A set-associative cache is characterized by the number of **banks** or **ways** it uses, giving an n-way set-associative cache.

A set is formed by all the blocks (one for each bank) that share the same index. Each set is implemented with a direct-mapped cache. A cache request is broadcast to all banks simultaneously. If any of the sets has the location, the cache reports a hit.

Although memory locations map onto blocks using the same function, there are n separate blocks for each set of locations. Therefore, we can simultaneously cache several locations that happen to map onto the same cache block. The set associative cache structure incurs a little extra overhead and is slightly slower than a direct-mapped cache, but the higher hit rates that it can provide often compensate.

The set-associative cache generally provides higher hit rates than the direct mapped cache because conflicts between a small number of locations can be resolved within the cache. The set-associative cache is somewhat slower, so the CPU designer has to be

careful that it doesn't slow down the CPU's cycle time too much. A more important problem with set-associative caches for embedded program.



Design is predictability. Because the time penalty for a cache miss is so severe, we often want to make sure that critical segments of our programs have good behaviour in the cache. It is relatively easy to determine when two memory locations will conflict in a direct-mapped cache.

Conflicts in a set-associative cache are more subtle, and so the behaviour of a set-associative cache is more difficult to analyze for both humans and programs

13. Explain the importance of CPU performance in embedded applications. (8) [CO1-L1]

Now that we have an understanding of the various types of instructions that CPUs can execute, we can move on to a topic particularly important in embedded computing: How fast can the CPU execute instructions? In this section, we consider three factors that can substantially influence program performance: pipelining and caching.

Pipelining

Modern CPUs are designed as **pipelined** machines in which several instructions are executed in parallel. Pipelining greatly increases the efficiency of the CPU. But like any pipeline, a CPU pipeline works best when its contents flow smoothly. Some sequences of instructions can disrupt the flow of information in the pipeline and, temporarily at least, slow down the operation of the CPU.

- **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. But since the pipeline has three stages, an instruction is completed in every clock cycle. In other words, the pipeline has a **throughput** of one instruction per cycle.

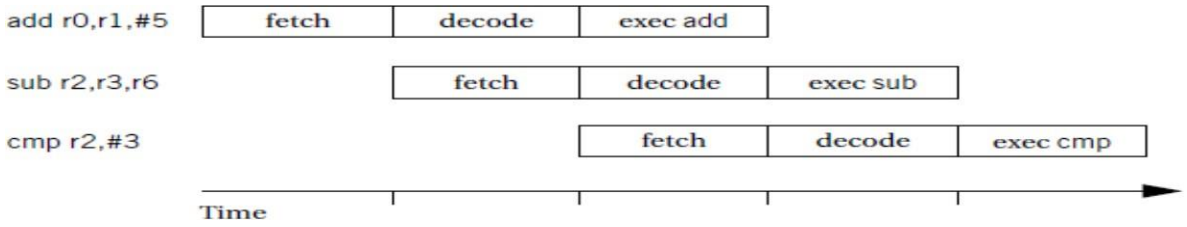
Figure illustrates the position of instructions in the pipeline during execution using the notation introduced by Hennessy and Patterson [Hen06]. A vertical slice through the timeline shows all instructions in the pipeline at that time. By following an instruction horizontally, we can see the progress of its execution.

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.

Caching

We have already discussed caches functionally. Although caches are invisible in the programming model, they have a profound effect on performance. We introduce caches because they substantially reduce memory access time when the requested location is in the cache.



However, the desired location is not always in the cache since it is considerably smaller than main memory. As a result, caches cause the time required to access memory to vary considerably. The extra time required to access a memory location not in the cache is often called the **cache miss penalty**.

The amount of variation depends on several factors in the system architecture, but a cache miss is often several clock cycles slower than a cache hit. The time required to access a memory location depends on whether the
requested location is in the cache. However, as we have seen, a location may not be in the cache for several reasons.

- At a compulsory miss, the location has not been referenced before.
- At a conflict miss, two particular memory locations are fighting for the same cache line.
- At a capacity miss, the program's working set is simply too large for the cache.

The contents of the cache can change considerably over the course of execution of a program. When we have several programs running concurrently on the CPU.

14. Describe the importance of CPU power consumption. (4) [CO1-L1]

Power consumption is, in some situations, as important as execution time. In this section we study the characteristics of CPUs that influence power consumption and mechanisms provided by CPUs to control how much power they consume. 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. Generally, we will

use the term power as shorthand for energy and power consumption, distinguishing between them only when necessary.

The high-level power consumption characteristics of CPUs and other system components are derived from the circuits used to build those components. Today, virtually all digital systems are built with **complementary metal oxide semiconductor (CMOS)** circuitry. The detailed circuit characteristics are best left to a study of VLSI design [Wol08], but 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.

Unit – II

Embedded computing platform design

Part – A

1. What is the bus protocols especially, the four-cycle handshake? [CO2-L1-April 2014]

Protocols are the set of rules and conditions for the data communication. The basic building block of most bus protocols is the four-cycle handshake.

Handshake ensures that when two devices want to communicate. One is ready to transmit and other is ready to receive.

The handshake uses a pair of wires dedicated to the handshake; such as enq(meaning enquiry) and ack (meaning acknowledge). Extra wires are used for the data transmitted during handshake.

2. What is a data flow graph? [CO2-L1-April 2014]

A data flow graph is a model of a program with no conditions. In a high level programming language, a code segment with no conditions and one entry point and exit point.

3. What are CPU buses? [CO2-L1-Nov/Dec 2013 & May/June 2013]

- ✓ Data bus
- ✓ Address bus
- ✓ Control bus
- ✓ System bus.

4. List out the various compilation techniques. [CO2-L1-Nov/Dec 2013]

There are three types of compilation techniques:

- ✓ Analysis and optimization of execution time.
- ✓ Power energy and program size
- ✓ Program validation and testing.

5. State the basic principles of basic compilation techniques. [CO2-L1-May/June 2013]

- ✓ Compilation combines translation and optimization.
- ✓ The high level language program is translated in to lower level form of instructions; optimizations try to generate better instruction sequences.
- ✓ Compilation = Translation + optimization

6. Name any two techniques used to optimize execution time of program. [CO2-L1-Nov/Dec 2012]

- ✓ Instruction level optimization
- ✓ Machine independent optimization.

7. What does a linker do? [CO2-L1-Nov/Dec 2012]

- ✓ A linker allows a program to be stitched together out of several smaller pieces.
- ✓ The linker operates on the object files created by the assembler and modifies the assemble code to make the necessary links between files.

8. What are the four types of data transfer in USB? [CO2-L1-May/June 2012]

- ✓ Control transfer
- ✓ Interrupt transfer
- ✓ Bulk transfer
- ✓ Isochronous transfer (sequence of data)

9. Give the limitation of polling techniques. [CO2-L1-May/June 2012]

- ✓ It is wasteful of the processors time, as it needlessly checks the status of all devices all the time.
- ✓ It is inherently slow, as it checks the status of all input/output devices before it comes back to check any given one again.
- ✓ Priority of the device cannot be determined frequently.

10. Define BUS. [CO2-L1]

A bus is a connection of wires. The bus defines a protocol by which the CPU communicates with memory and I/O devices.

11. Define memory mapped I/O? [CO2-L1]

Memory-Mapped I/O (MMIO) and Port-Mapped I/O (PMIO) (which is also called isolated i/o) are two complementary methods of performing input/output between the cpu and peripheral devices in a computer. Memory-mapped I/O, uses the same address bus to address both memory and i/o devices – the memory and registers of the i/o devices are mapped to (associated with) address values. So when an address is accessed by the cpu, it may refer to a portion of physical ram, but it can also refer to memory of the i/o device.

12. Define RAM? [CO2-L1]

RAM is an acronym for random access memory, a type of computer memory that can be accessed randomly; that is, any byte of memory can be accessed without touching the preceding bytes. RAM is the most common type of memory found in computers and other devices, such as printers.

13. What is dynamic RAM? [CO2-L1]

Dynamic Random-Access Memory (DRAM) is a type of random-access memory that stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can be either charged or discharged; these two states are taken to represent the two values of a bit, conventionally called 0 and 1.

14. What is ROM? [CO2-L1]

Read-Only memory (ROM) is a class of storage medium used in computers and other electronic devices. Data stored in ROM can only be modified slowly, with difficulty, or not at all, so it is mainly used to distribute firmware (software that is very closely tied to specific hardware, and unlikely to need frequent updates).

15. What are the 4 types of data transfer used in USB? [CO2-L1]

Control Transfer, Isochronous Transfer, Interrupt Transfer, Bulk Transfer 19. Give the limitations of polling technique.

1) it is wasteful of the processors time, as it needlessly checks the status of all devices all the time,2) it is inherently slow, as it checks the status of all I/O devices before it comes back to check any given one again,3) when fast devices are connected to a system, polling may simply not be fast enough to satisfy the minimum service requirements, 4) priority of the device is determined by the order in the polling loop, but it is possible to change it via software.

16. What is USB? Where is it used? [CO2-L1]

It is an external bus standard that supports data transfer rates of 12 Mbps. A single USB port can be used to connect up to 127 peripheral devices, such as mice, modems, and keyboards.

Part - B

1. Explain about various CPU BUS configurations in embedded systems. (8)[CO2-L1]

A computer system encompasses much more than the CPU;it also includes memory and I/O devices. The **bus** is the mechanism by which the CPU communicates with memory and devices. A bus is, at a minimum, a collection of wires, but the bus also defines a protocol by which the CPU, memory, and devices communicate. One of the major roles of the bus is to provide an interface to memory. (Of course, I/O devices also connect to the bus.)

Bus Protocols:

The basic building block of most bus protocols is the **four-cycle handshake**, illustrated in Figure 2.1. The handshake ensures that when two devices want to communicate, one is ready to transmit and the other is ready to receive.

The handshake uses a pair of wires dedicated to the handshake: **enq** (meaning enquiry) and **ack** (meaning acknowledge). Extra wires are used for the data transmitted during the handshake. The four cycles are described below.

1. Device 1 raises its output to signal an enquiry, which tells device 2 that it should get ready to listen for data.

2. When device 2 is ready to receive, it raises its output to signal an acknowledgment. At this point, devices 1 and 2 can transmit or receive.

3. Once the data transfer is complete, device 2 lowers its output, signaling that it has received the data.

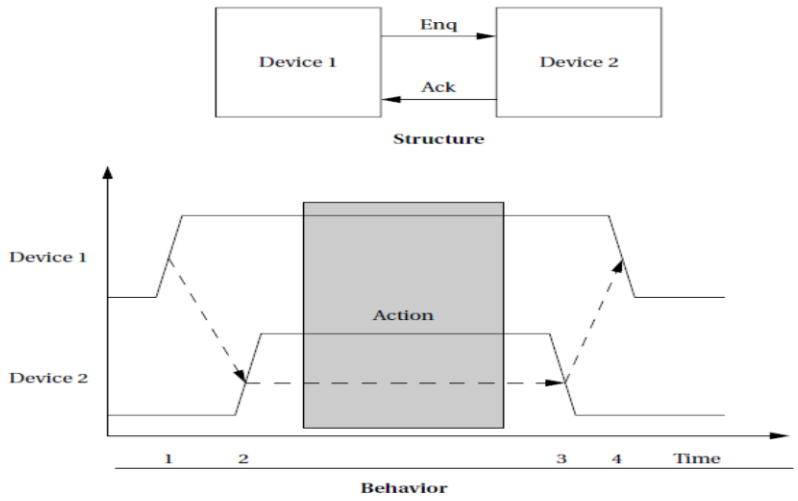
4. After seeing that ack has been released, device 1 lowers its output.

At the end of the handshake, both handshaking signals are low, just as they were at the start of the handshake. The system has thus returned to its original state in readiness for another handshake-enabled data transfer.

Microprocessor buses build on the handshake for communication between the CPU and other system components. The term bus is used in two ways.

The most basic use is as a set of related wires, such as address wires. However, the term may also mean a protocol for communicating between components.

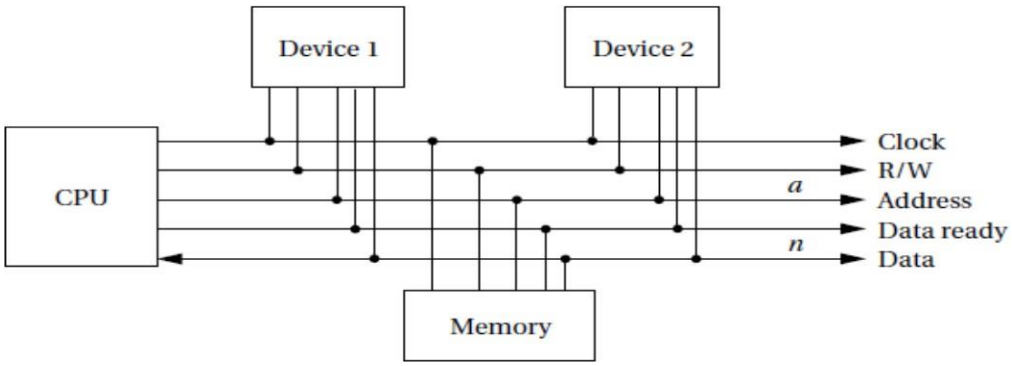
To avoid confusion, we will use the term **bundle** to refer to a set of related signals. The fundamental bus operations are reading and writing. Figure 2.2 shows the structure of a typical bus that supports reads and writes.



The four-cycle handshake

The major components follow:

- Clock provides synchronization to the bus components,
- R/W is true when the bus is reading and false when the bus is writing,
- Address is an a-bit bundle of signals that transmits the address for an access,
- Data is an n-bit bundle of signals that can carry data to or from the CPU, and
- Data ready signals when the values on the data bundle are valid.

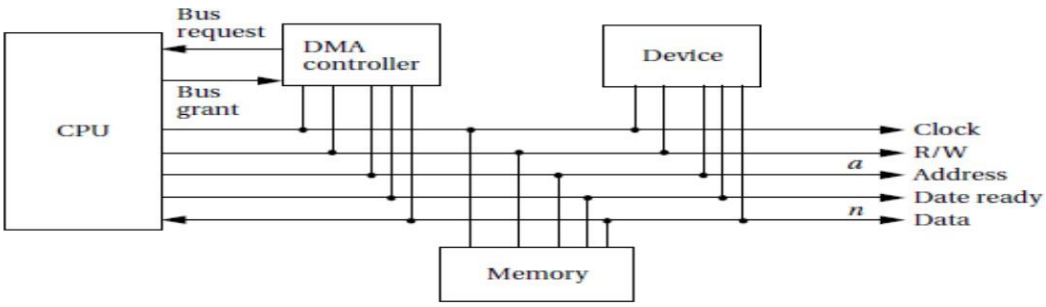


A typical microprocessor bus.

DMA:

Standard bus transactions require the CPU to be in the middle of every read and write transaction. However, there are certain types of data transfers in which the CPU does not need to be involved.

For example, a high-speed I/O device may want to transfer a block of data into memory. While it is possible to write a program that alternately reads the device and writes to memory, it would be faster to eliminate the CPU's involvement and let the device and memory communicate directly. This capability requires that some unit other than the CPU be able to control operations on the bus.



A bus with a DMA controller.

Direct memory access (DMA) is a bus operation that allows reads and writes not controlled by the CPU. A DMA transfer is controlled by a **DMA controller**, which requests control of the bus from the CPU.

After gaining control, the DMA controller performs read and write operations directly between devices and memory. Figure 2.3 shows the configuration of a bus with a DMA controller. The DMA requires the CPU to provide two additional bus signals:

- The **bus request** is an input to the CPU through which DMA controllers ask for ownership of the bus.
- The **bus grant** signals that the bus has been granted to the DMA controller.

A device that can initiate its own bus transfer is known as a **bus master**. Devices that do not have the capability to be **bus masters** do not need to connect to a bus request and bus grant.

The DMA controller uses these two signals to gain control of the bus using a classic four-cycle handshake. The bus request is asserted by the DMA controller when it wants to control the bus, and the bus grant is asserted by the CPU when the bus is ready.

The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller. When it does grant control, it stops driving the other bus signals: R/W, address, and so on. Upon becoming bus master, the DMA controller has control of all bus signals (except, of course, for bus request and bus grant).

2. Describe the memory devices used in embedded system design. (8) [CO2-L1]

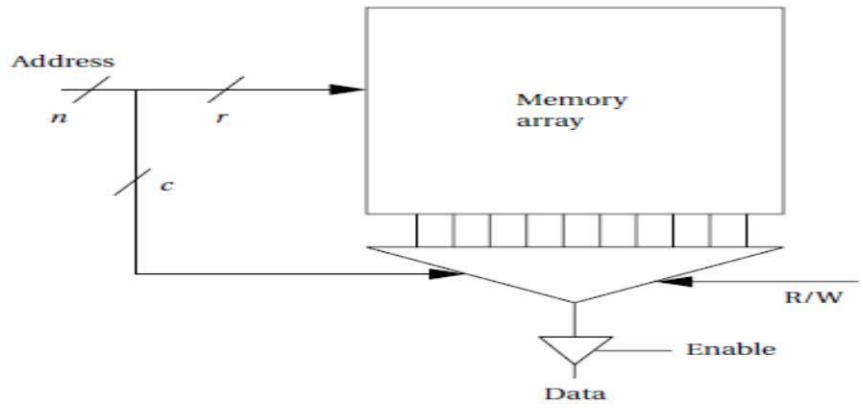
There are several varieties of both read-only and read/write memories, each with its own advantages. After discussing some basic characteristics of memories, we describe RAMs and then ROMs.

Memory Device Organization

The most basic way to characterize a memory is by its capacity, such as 256 MB. However, manufacturers usually make several versions of a memory of a given size, each with a different data width. For example, a 256-MB memory may be available in two versions:

- As a 64M *4-bit array, a single memory access obtains an 8-bit data item, with a maximum of 226 different addresses.
- As a 32 M* 8-bit array, a single memory access obtains a 1-bit data item, with a maximum of 223 different addresses.

The height/width ratio of a memory is known as its **aspect ratio**. The best aspect ratio depends on the amount of memory required. Internally, the data are stored in a two-dimensional array of memory cells as shown in Figure 2.4. Because the array is stored in two dimensions, the n-bit address received by the chip is split into a row and a column address (with n =r+ c).



Internal organization of a memory device

The row and column select a particular memory cell. If the memory's external width is 1 bit, the column address selects a single bit; for wider data widths, the column address can be used to select a subset of the columns. Most memories include an enable signal that controls the tri-stating of data onto the memory's pins.

Random-Access Memories:

Random-access memories can be both read and written. They are called random access because, unlike magnetic disks, addresses can be read in any order. Most bulk memory in modern systems is **dynamic RAM (DRAM)**. DRAM is very dense; it does,

however, require that its values be **refreshed** periodically since the values inside the memory cells decay over time.

The dominant form of dynamic RAM today is the **synchronous DRAMs (SDRAMs)**, which uses clocks to improve DRAM performance. SDRAMs use Row Address Select (RAS) and Column Address Select (CAS) signals to break the address into two parts, which select the proper row and column in the RAM array. Signal transitions are relative to the SDRAM clock,which allows the internal SDRAM operations to be pipelined.

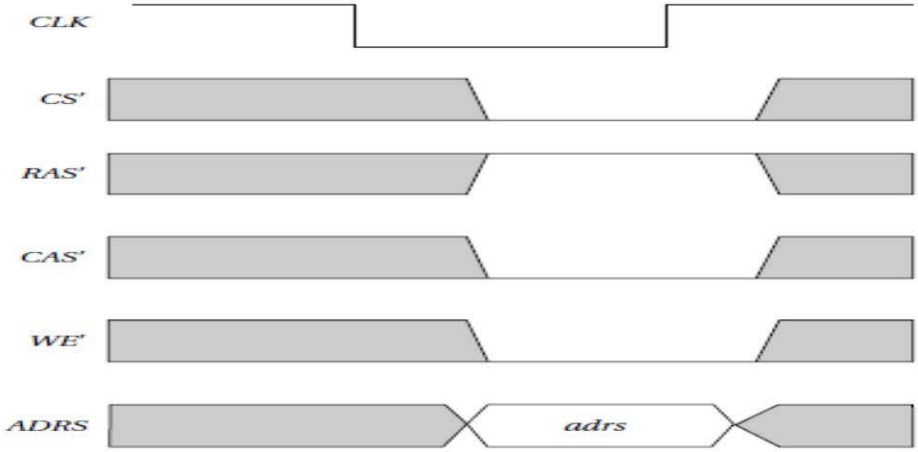


Fig. Timing diagram for a read on a synchronous DRAM

As shown in Figure 2.5, transitions on the control signals are related to a clock [Mic00]. RAS_ and CAS_ can therefore become valid at the same time.

The address lines are not shown in full detail here; some address lines may not be active depending on the mode in use. SDRAMs use a separate refresh signal to control refreshing. DRAM has to be refreshed roughly once per millisecond.

Rather than refresh the entire memory at once, DRAMs refresh part of the memory at a time. When a section of memory is being refreshed, it cannot be accessed until the refresh is complete. The memory refresh occurs over fairly few seconds so that each section is refreshed every few microseconds.

SDRAMs include registers that control the mode in which the SDRAM operates. SDRAMs support burst modes that allow several sequential addresses to be accessed by sending only one address. SDRAMs generally also support an interleaved mode that exchanges pairs of bytes.

Read-Only Memories:

Read-only memories (ROMs) are preprogrammed with fixed data. They are very useful in embedded systems since a great deal of the code, and perhaps some data, does not change over time. Read-only memories are also less sensitive to radiation induced errors.

There are several varieties of ROM available. The first-level distinction to be made is between **factory-programmed ROM** (sometimes called **mask-programmed ROM**) and **field-programmable ROM**.

Factory-programmed ROMs are ordered from the factory with particular programming. ROMs can typically be ordered in lots of a few thousand, but clearly factory programming is useful only when the ROMs are to be installed in some quantity.

Field-programmable ROMs, on the other hand, can be programmed in the lab. **Flash memory** is the dominant form of field-programmable ROM and is electrically erasable.

Flash memory uses standard system voltage for erasing and programming, allowing it to be reprogrammed inside a typical system. This allows applications such as automatic distribution of upgrades—the flash memory can be reprogrammed while downloading the new memory contents from a telephone line.

Early flash memories had to be erased in their entirety; modern devices allow memory to be erased in blocks. Most flash memories today allow certain blocks to be protected.

A common application is to keep the boot-up code in a protected block but allow updates to other memory blocks on the device. As a result, this form of flash is commonly known as **boot-block flash**.

3. Explain the different I/O devices used in embedded system. (16) [CO2-L1]

Some of these devices are often found as on-chip devices in micro-controllers; others are generally implemented separately but are still commonly used. Looking at a few important devices now will help us understand both the requirements of device interfacing.

Timers and Counters:

Timers and **counters** are distinguished from one another largely by their use, not their logic. Both are built from adder logic with registers to hold the current value, with an increment input that adds one to the current register value.

However, a timer has its count connected to a periodic clock signal to measure time intervals, while a counter has its count input connected to an aperiodic signal in order to count the number of occurrences of some external event. Because the same logic can be used for either purpose, the device is often called a **counter/timer**.

Figure shows enough of the internals of a counter/timer to illustrate its operation. An n-bit counter/timer uses an n-bit register to store the current state of the count and an array of half subtractors to decrement the count when the count signal is asserted.

Combinational logic checks when the count equals zero; the done output signals the zero count. It is often useful to be able to control the time-out, rather than require exactly 2n events to occur. For this purpose, a reset register provides the value with which the count register is to be loaded.

The counter/timer provides logic to load the reset register. Most counters provide both cyclic and acyclic modes of operation. In the cyclic mode, once the counter reaches the done state, it is automatically reloaded and the counting process continues. In acyclic mode, the counter/timer waits for an explicit signal from the microprocessor to resume counting.

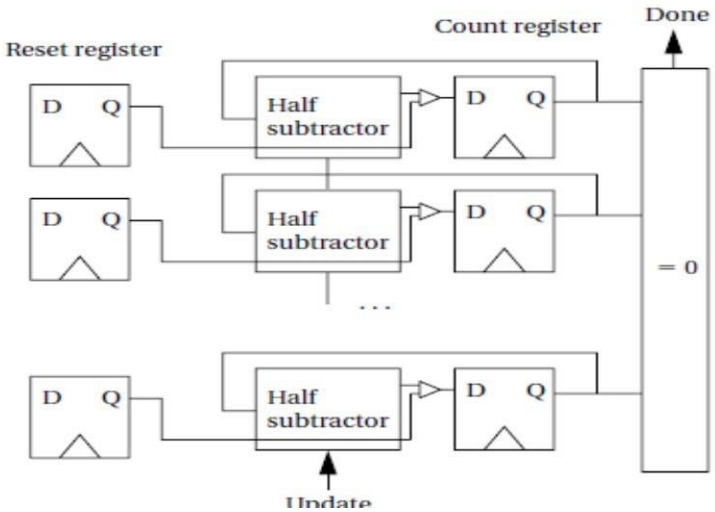


Fig. Internals of a counter/timer

A **watchdog timer** is an I/O device that is used for internal operation of a system. As shown in Figure 2.7, the watchdog timer is connected into the CPU bus and also to the CPU's reset line.

The CPU's software is designed to periodically reset the watchdog timer, before the timer ever reaches its time-out limit. If the watchdog timer ever does reach that limit, its time-out action is to reset the processor. In that case, the presumption is that either a software flaw or hardware problem has caused the CPU to misbehave. Rather than diagnose the problem, the system is reset to get it operational as quickly as possible.

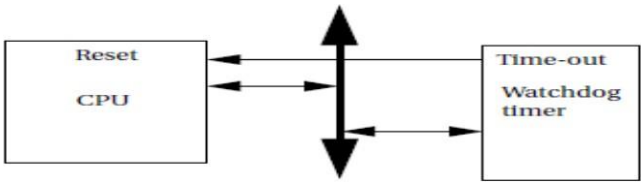


Fig. A Watchdog timer

A/D and D/A Converters

Analog/digital (A/D) and **digital/analog (D/A)** converters (typically known as **ADCs** and **DACs**, respectively) are often used to interface non digital devices to embedded systems.

The design of A/D and D/A converters themselves is beyond the scope of this book; we concentrate instead on the interface to the microprocessor bus. Because A/D

conversion requires more complex circuitry, it requires a somewhat more complex interface.

Analog/digital conversion requires sampling the analog input before converting it to digital form. A control signal causes the A/D converter to take a sample and digitize it.

There are several different types of A/D converter circuits, some of which take a constant amount of time, while the conversion time of others depends on the sampled value. Variable-time converters provide a done signal so that the microprocessor knows when the value is ready.

A typical A/D interface has, in addition to its analog inputs, two major digital inputs. A data port allows A/D registers to be read and written, and a clock input tells when to start the next conversion.

D/A conversion is relatively simple, so the D/A converter interface generally includes only the data value. The input value is continuously converted to analog form.

LEDs

Light-emitting diodes (LEDs) are often used as simple displays by themselves, and arrays of LEDs may form the basis of more complex displays. Figure 2.8 shows how to connect an LED to a digital output.

A resistor is connected between the output pin and the LED to absorb the voltage difference between the digital output voltage and the 0.7 V drop across the LED. When the digital output goes to 0, the LED voltage is in the devices off region and the LED is not on.

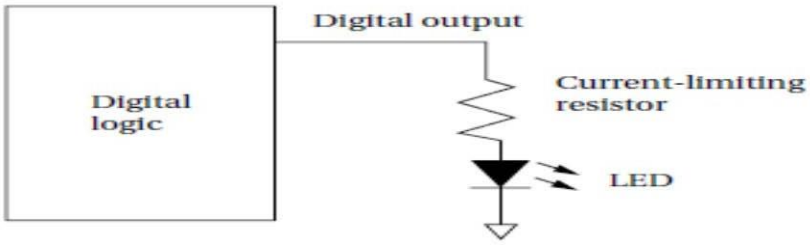


Fig.An LED connected to a digital output.

Displays

A display device may be either directly driven or driven from a frame buffer. Typically, displays with a small number of elements are driven directly by logic, while large displays use a RAM frame buffer.

The n-digit array, shown in Figure 2.9, is a simple example of a display that is usually directly driven. A single-digit display typically consists of seven segments; each segment may be either an LED or a **liquid crystal display (LCD)** element.

Display relies on the digits being visible for some time after the drive to the digit is removed, which is true for both LEDs and LCDs.

The digit input is used to choose which digit is currently being updated, and the selected digit activates its display elements based on the current data value.

The display's driver is responsible for repeatedly scanning through the digits and presenting the current value of each to the display.

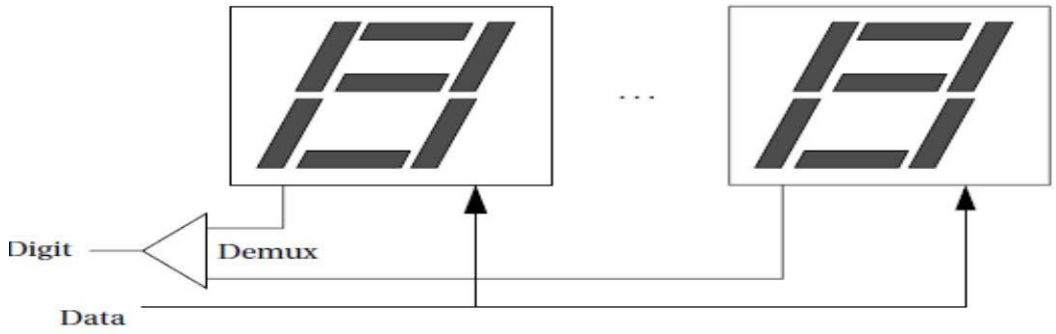


Fig. An n-digit display

A **frame buffer** is a RAM that is attached to the system bus. The microprocessor writes values into the frame buffer in whatever order is desired.

The pixels in the frame buffer are generally written to the display in **raster order** (by tradition, the screen is in the fourth quadrant) by reading pixels sequentially.

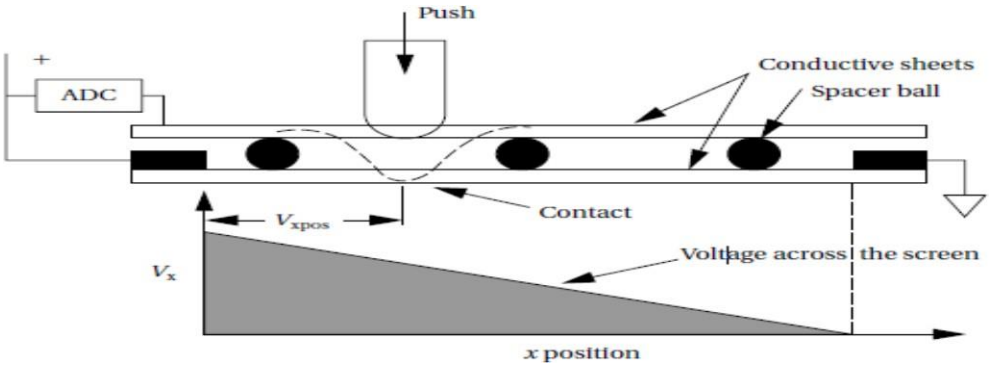
Many large displays are built using LCD. Each pixel in the display is formed by a single liquid crystal.

Early LCD panels were called **passive matrix** because they relied on a two-dimensional grid of wires to address the pixels. Modern LCD panels use an **active matrix** system that puts a transistor at each pixel to control access to the LCD. Active matrix displays provide higher contrast and a higher-quality display.

Touchscreens

A **touchscreen** is an input device overlaid on an output device. The touchscreen registers the position of a touch to its surface. By overlaying this on a display, the user can react to information shown on the display.

The two most common types of touchscreens are resistive and capacitive. A resistive touchscreen uses a two-dimensional voltmeter to sense position. As shown in Figure 2.10, the touchscreen consists of two conductive sheets separated by spacer balls. The top conductive sheet is flexible so that it can be pressed to touch the bottom sheet.



A voltage is applied across the sheet; its resistance causes a voltage gradient to appear across the sheet. The top sheet samples the conductive sheet's applied voltage at the contact point. An analog/digital converter is used to measure the voltage and resulting position.

The touchscreen alternates between x and y position sensing by alternately applying horizontal and vertical voltage gradients.

4. Write short notes on component interfacing. (5) [CO2-L1]

Building the logic to interface a device to a bus is not too difficult but does take some attention to detail. We first consider interfacing memory components to the bus, since that is relatively simple, and then use those concepts to interface to other types of devices.

Memory Interfacing

If we can buy a memory of the exact size we need, then the memory structure is simple. If we need more memory than we can buy in a single chip, then we must construct the memory out of several chips. We may also want to build a memory that is wider than we can buy on a single chip; for example, we cannot generally buy a 32-bit-wide memory chip. We can easily construct a memory of a given width (32 bits, 64 bits, etc.) by placing RAMs in parallel.
We also need logic to turn the bus signals into the appropriate memory signals. For example, most busses won't send address signals in row and column form. We also need to generate the appropriate refresh signals.

Device Interfacing

Some I/O devices are designed to interface directly to a particular bus, forming **glueless interfaces**. But **glue logic** is required when a device is connected to a bus for which it is not designed.
An I/O device typically requires a much smaller range of addresses than a memory, so addresses must be decoded much more finely. Some additional logic is required to cause the bus to read and write the device's registers.

5. Discus the steps involved while designing with computing platforms. (12) [CO2-L1]

System Architecture

We know that an architecture is a set of elements and the relationships between them that together form a single unit. The architecture of an embedded computing system is the blueprint for implementing that system—it tells you what components you need and how you put them together.

The architecture of an embedded computing system includes both hardware and software elements. Let's consider each in turn.

The hardware architecture of an embedded computing system is the more obvious manifestation of the architecture since you can touch it and feel it. It includes several elements, some of which may be less obvious than others.

CPU An embedded computing system clearly contains a microprocessor. But which one? There are many different architectures, and even within an architecture we can select between models that vary in clock speed, bus data width, integrated peripherals, and so on. The choice of the CPU is one of the most important, but it cannot be made without considering the software that will execute on the machine.

Bus The choice of a bus is closely tied to that of a CPU, since the bus is an integral part of the microprocessor. But in applications that make intensive use of the bus due to I/O or other data traffic, the bus may be more of a limiting factor than the CPU. Attention must be paid to the required data bandwidths to be sure that the bus can handle the traffic.

Memory Once again, the question is not whether the system will have memory but the characteristics of that memory. The most obvious characteristic is total size, which depends on both the required data volume and the size of the program instructions. The ratio of ROM to RAM and selection of DRAM versus SRAM can have a significant influence on the cost of the system. The speed of the memory will play a large part in determining system performance.

Input and output devices The user's view of the input and output mechanisms may not correspond to the devices connected to the microprocessor. For example, a set of switches and knobs on a front panel may all be controlled by a single microcontroller, which is in turn connected to the main CPU.

For a given function, there may be several different devices of varying sophistication and cost that can do the job. The difficulty of using a particular device, such as the amount of glue logic required to interface it, may also play a role in final device selection.

Hardware Design

The design complexity of the hardware platform can vary greatly, from a totally off-the-shelf solution to a highly customized design.

At the board level, the first step is to consider **evaluation boards** supplied by the microprocessor manufacturer or another company working in collaboration with the manufacturer. Evaluation boards are sold for many microprocessor systems; they typically include the CPU, some memory, a serial link for downloading programs, and some minimal number of I/O devices. Figure 2.11 shows an ARM evaluation board manufactured by Sharp.

The evaluation board may be a complete solution or provide what you need with only slight modifications. If the evaluation board is supplied by the microprocessor vendor, its design (netlist, board layout, etc.) may be available from the vendor; companies provide such information to make it easy for customers to use their microprocessors. If the evaluation board comes from a third party, it may be possible to contract them to design a new board with your required modifications, or you can start from scratch on a new board design.

The other major task is the choice of memory and peripheral components. In the case of I/O devices, there are two alternatives for each device: selecting a component from a catalog or designing one yourself. When shopping for devices from a catalog, it is important to read data sheets carefully it may not be trivial to figure out whether the device does what you need it to do.

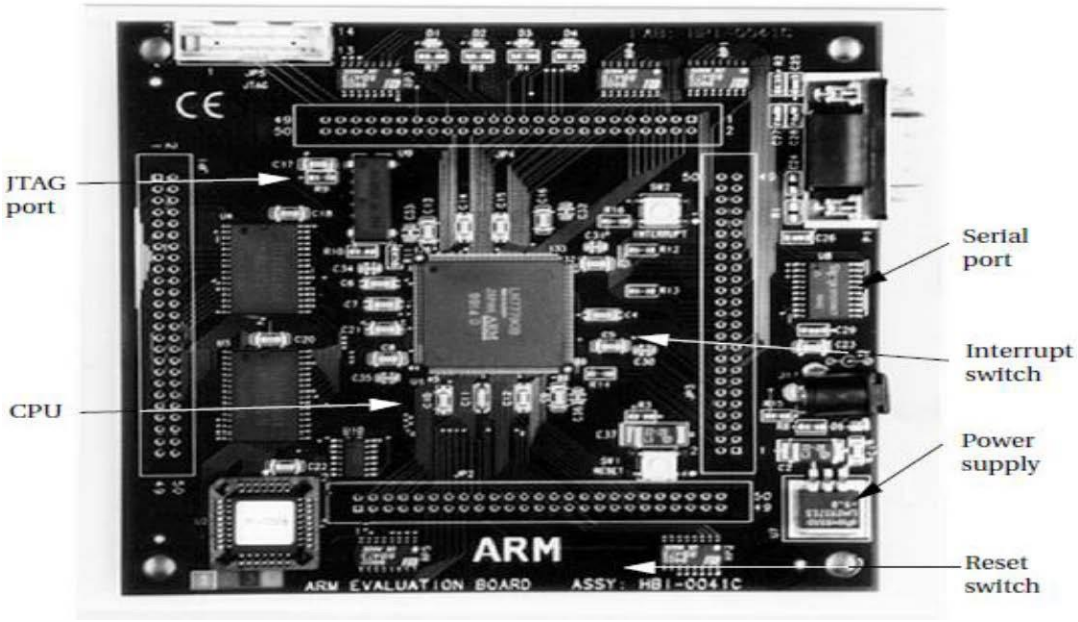


Fig. An ARM evaluation board

.

Development Environments

A typical embedded computing system has a relatively small amount of everything, including CPU horsepower, memory, I/O devices, and so forth. As a result, it is common to do at least part of the software development on a PC or workstation known as a **host** as illustrated in Figure The hardware on which the code will finally run is known as the **target**. The host and target are frequently connected by a USB link, but a higher-speed link such as Ethernet can also be used.

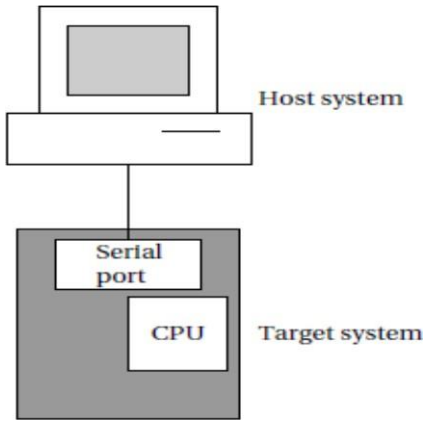


Fig.Connecting a host and a target system

The target must include a small amount of software to talk to the host system. That software will take up some memory, interrupt vectors, and so on, but it should generally leave the smallest possible footprint in the target to avoid interfering with the application software.
The host should be able to do the following:

- load programs into the target,
- start and stop program execution on the target, and
- examine memory and CPU registers

Debugging Techniques:

A good deal of software debugging can be done by compiling and executing the code on a PC or workstation. But at some point it inevitably becomes necessary to run code on the embedded hardware platform.

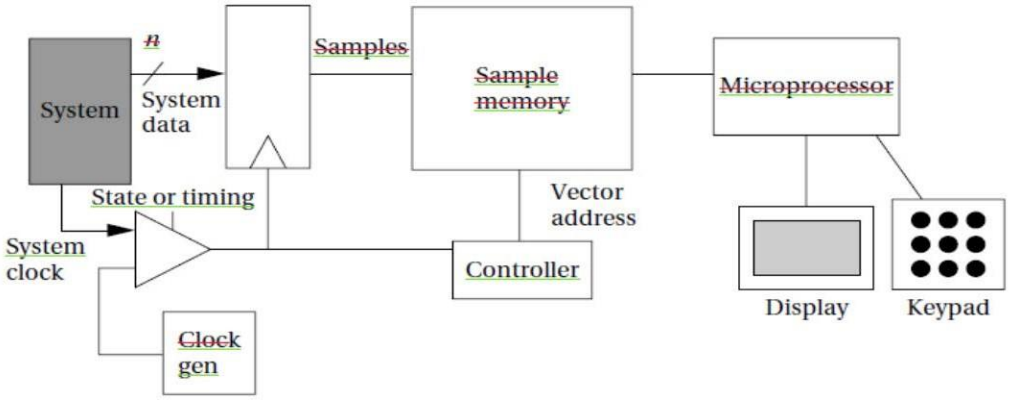
Embedded systems are usually less friendly programming environments than PCs. Nonetheless, the resourceful designer has several options available for debugging the system.

The serial port found on most evaluation boards is one of the most important debugging tools. In fact, it is often a good idea to design a serial port into an embedded system even if it will not be used in the final product; the serial port can be used not only for development debugging but also for diagnosing problems in the field.

Another very important debugging tool is the **breakpoint**. The simplest form of a breakpoint is for the user to specify an address at which the program's execution is to break. When the PC reaches that address, control is returned to the monitor program. From the monitor program, the user can examine and/or modify CPU registers, after which execution can be continued. Implementing breakpoints does not require using exceptions or external devices.

Debugging Challenges

Logical errors in software can be hard to track down, but errors in real-time code can create problems that are even harder to diagnose. Real-time programs are required to finish their work within a certain amount of time; if they run too long, they can create very unexpected behavior.



The exact results of missing real-time deadlines depend on the detailed characteristics of the I/O devices and the nature of the timing violation. This makes debugging real-time problems especially difficult.

Unfortunately, the best advice is that if a system exhibits truly unusual behavior, missed deadlines should be suspected. In-circuit emulators, logic analyzers, and even LEDs can be useful tools in checking the execution time of real-time code to determine whether it in fact meets its deadline.

6. Explain about consumer electronics architecture. (6) [CO2-L1]

Although some predict the complete convergence of al consumer electronic functions into a single device, much as has happened to the personal computer, we still have a variety of devices with different functions. However, consumer electronics devices have converged over the past decade around a set of common features that are supported by common architectural features. Not all devices have all features, depending on the way the device is to be used, but most devices select feature from a common menu. Similarly, there is no single platform for consumer electronics devices, but the architectures in use are organized around some common themes,

This convergence is possible because these devices implement a few basic types of function in various communications: multimedia and communications. The style of multimedia or communications may vary, and different devices may use different formats, but this causes variations in hardware and software components within the basic architectural templates. In this section we will look at general features of consumer electronics devices; in the following section we will look at general features of consumer electronics devices; in the following sections we will study a few devices in more detail.

Consumer electronics devices provide several types of services in different combinations:

Multimedia: The media may be audio, still images, or video (which includes both motion pictures and audio). These multimedia objects are generally stored in compressed form and must be uncompressed to be played (audio playback, video viewing, etc.). A large and growing number of standards have been developed for multimedia compression: MP3, Dolby Digital, and so on for audio; JPEG for still images; MPEG-2, MPEG-4, H.264, and so on for video.

Data storage and management: Because people want to select what multimedia objects they save or play, data storage goes hand in hand with multimedia capture and display. Many devices provide Pc compatible file systems so that data can be shared more easily.

Communications: Communications may be relatively simple, such as USB interface to a host computer. The communications link may also be more sophisticated, such as an Ethernet port or a cellular telephone link.

Consumer electronics devices must meet several types of strict nonfunctional requirements as well. Many devices are battery operated, which means that they must operate under strict energy budgets, A typical battery for a portable device provides only about 75mW, which must support not only the processors and digital electronics but also the display, radio, and so on. Consumer electronics must also be very inexpensive. A typical primary processing chip must sell in the neighborhood of $10

Let’s consider some basic use cases of some basic operations fig shows a use case for selection and playing a multimedia object. Selection an object makes use of both the user interface and the file system playing also makes use of the file system as well as the decoding subsystem and i/o subsystem,

The connection may be either over a local connection like USB or over the internet. While some operations may be performed locally on the client device, most of the work is done on the host system while the connection is established.

Hardware architectures

storage system provides bulk permanent storage the network interface may provide a simple sub connection or a full blown internet connection

Multiprocessor architectures are common in many consumer multimedia devices Dsp and CPU may be added the RICS CPU runs the operate system runs the user interface maintains the file system and so on the dsp performs signal processing the dsp may be programmable in some systems in other cases it may be one or more hardwired accelerators.

Operating systems

The operating system that runs on the CPU must maintain processes and the file system processes are necessary to provide concurrency for example the user wants to be able to push a button while the device is playing back audio Depending on to complexity of the device the operation system may not need to create tasks dynamically if all tasks can be created using initialization code, the operation system can be made smaller and simpler

7. Analyze the platform level performance in an embedded system. (8) [CO2-L1]

Bus based systems add another layer of complication to performance analysis platform level performance involve much more than the CPU we often focus on the CPU because it processes instructions but any part of the system can affect total system performance. More precisely the CPU provides an upper bound on performance but any other part of the system can slow down the CPU merely counting instruction execution times is not enough

Consider the simple system we want to move data from memory to the CPU to process it to get the data from memory to the CPU we must

- Read from the memory
- Transfer over the bus to the cache
- Transfer from the cache to the CPU

The time required to transfer from the cache to the CPU is included in the instruction execution time , but the other two times are not .

Bandwidth as performance

The most basic measure of performance we are interested in is bandwidth the rate at which we can move data ultimately if we are interested in real time performance we are interested in real time performance measured in seconds but often the simplest way to measure performance is in units of clock cycles however different parts of the system will run at different clock rates. We have to make sure that we apply the right clock rate to each part of the performance estimate when we convert from clock cycles to seconds

Bus bandwidth

Bandwidth questions often come up when we are transferring large blocks of data for simplicity let's start by considering the bandwidth provided by only one system component the bus consider and image of 320 pixels with each pixel composed of 3 bytes of data this gives a grand total of 230 400 bytes of data if these images are video frames, we want to check if we can push one frame through the system within the 1/30 sec that we have to process a frame before the next one arrives.

Let us assume that we can transfer one byte of data every microsecond which implies a bus speed of 1 Mhz. in this case we would require 230400 us =0.23 sec to transfer one frame that is more than the 0.033 sec allotted to the data transfer we would have to increase the transfer rate by 7xto satisfy our performance requirement

We can increase bandwidth in two ways we can increase the clock rate of the bus or we can increase the amount of data transferred per clock cycle for example if we increased the bus to carry four bytes or 32 bits per transfer we would reduce the transfer time to 0.058 sec if we could also increase the bus clock rate to 2 Mhz. then we would reduce the transfer time to 0.029sec , which is within our time budget for the transfer

8. Explain the various components used in embedded programs. (8) [CO2-L1]

In this section, we consider code for three structures or components that are commonly used in embedded software: the state machine, the circular buffer, and the queue. State

machines are well suited to **reactive systems** such as user interfaces; circular buffers and queues are useful in digital signal processing.

State Machines

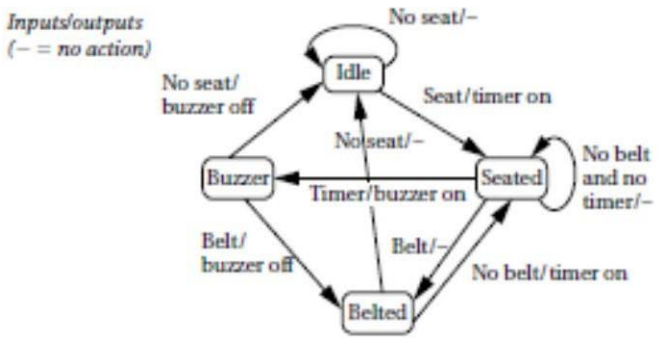
When inputs appear intermittently rather than as periodic samples, it is often convenient to think of the system as reacting to those inputs.

The reaction of most systems can be characterized in terms of the input received and the current state of the system. This leads naturally to a **finite-state machine** style of describing the reactive system's behaviour.

Moreover, if the behaviour is specified in that way, it is natural to write the program implementing that behaviour in a state machine style.

The state machine style of programming is also an efficient implementation of such computations. Finite-state machines are usually first encountered in the context of hardware
design.

A software state machine



Stream-Oriented Programming and Circular Buffers

The data stream style makes sense for data that comes in regularly and must be processed on the fly. For each sample, the filter must emit one output that depends on the values of the last n inputs. In a typical workstation application, we would process the samples over a given interval by reading them all in from a file and then computing the results all at once in a batch process. In an embedded system we must not only emit outputs in real time, but we must also do so using a minimum amount of memory.

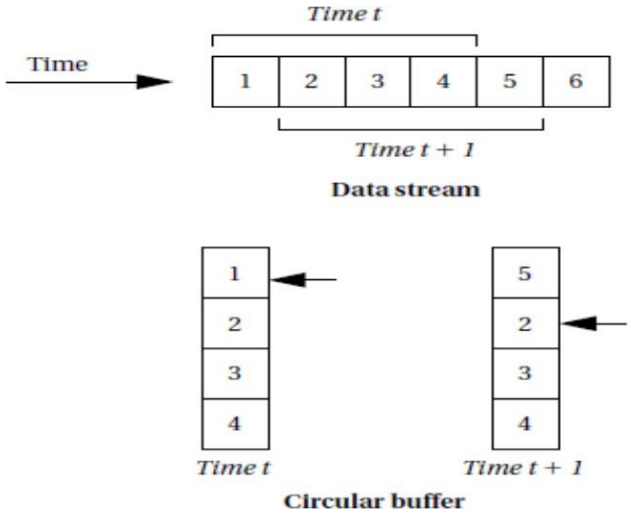


Fig. A circular buffer for streaming data

The circular buffer is a data structure that lets us handle streaming data in an efficient way. Figure illustrates how a circular buffer stores a subset of the data stream. At each point in time, the algorithm needs a subset of the data stream that forms a window into the stream. The window slides with time as we throw out old values no longer needed and add new values. Since the size of the window does not change, we can use a fixed-size buffer to hold the current data.
To avoid constantly copying data within the buffer, we will move the head of the buffer in time. The buffer points to the location at which the next sample will be placed; every time we add a sample, we automatically overwrite the oldest sample, which is the one that needs to be thrown out. When the pointer gets to the end of the buffer, it wraps around to the top.

9. Explain the different models of programs available for embedded system design. (6) [CO2-L1]

Data Flow Graphs:

A **data flow graph** is a model of a program with no conditionals. In a high-level programming language, a code segment with no conditionals—more precisely, with only one entry and exit point is known as a basic block. Figure 2.14 shows a simple basic block. As the C code is executed, we would enter this basic block at the beginning and execute all the statements.

```
w = a+b;
x = a-c;
y = x+d;
x = a+c;
z = y+e;
```

A basic block in C.

```
w = a+b;
x = a-c;
```

y = x1+d;
x2= a+c;
z = y+e;

The basic block in single-assignment form

Before we are able to draw the data flow graph for this code we need to modify it slightly. There are two assignments to the variable x—it appears twice on the left side of an assignment. We need to rewrite the code in **single-assignment form**, in which a variable appears only once on the left side.
Since our specification is C code, we assume that the statements are executed sequentially, so that any use of a variable refers to its latest assigned value. In this case, x is not reused in this block (presumably it is used elsewhere), so we just have to eliminate the multiple assignment to x. The result is shown in Figure 2.14, where we have used the names x1 and x2 to distinguish the separate uses of x.
The single-assignment form is important because it allows us to identify a unique location in the code where each named location is computed. As an introduction to the data flow graph, we use two types of nodes in the graph round nodes denote operators and square nodes represent values.
The value nodes may be either inputs to the basic block, such as a and b, or variables assigned to within the block, such as w and x1.

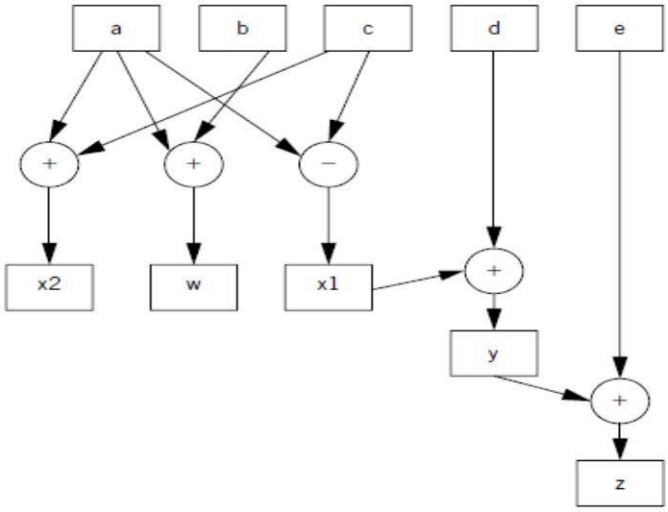


Fig.An extended data flow graph for our sample basic block

The data flow graph for our single-assignment code is shown in Figure 2.15.The single-assignment form means that the data flow graph is acyclic—if we assigned to x multiple times, then the second assignment would form a cycle in the graph including x and the operators used to compute x.

10. Explain the following design tools used in embedded system design. Assembly, linking and loading. (10) [CO2-L2]

Assembly and linking are the last steps in the compilation process they turn a list of instructions into an image of the program's bits in memory. Loading actually puts the program in memory so that it can be executed. In this section, we survey the basic

techniques required for assembly linking to help us understand the complete compilation and loading process.

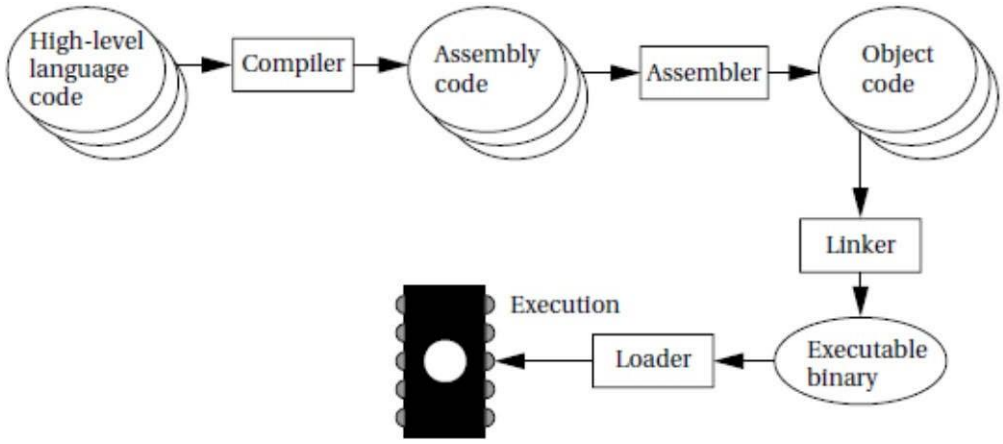


Fig. Program generation from compilation through loading.

Figure highlights the role of assemblers and linkers in the compilation process. This process is often hidden from us by compilation commands that do everything required to generate an executable program. As the figure shows, most compilers do not directly generate machine code, but instead create the instruction-level program in the form of human-readable assembly language. Generating assembly language rather than binary instructions frees the compiler writer from details extraneous to the compilation process, which includes the instruction format as well as the exact addresses of instructions and data.

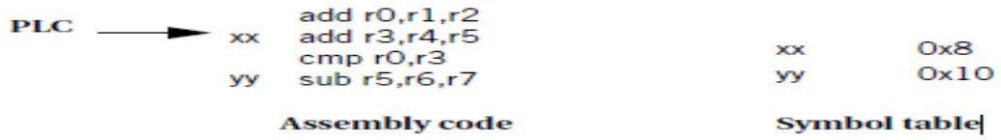
The assembler's job is to translate symbolic assembly language statements into bit-level representations of instructions known as **object code**. The assembler takes care of instruction formats and does part of the job of translating labels into addresses. However, since the program may be built from many files, the final steps in determining the addresses of instructions and data are performed by the linker, which produces an **executable binary** file. That file may not necessarily be located in the CPU's memory, however, unless the linker happens to create the executable directly in RAM. The program that brings the program into memory for execution is called a **loader**.

The simplest form of the assembler assumes that the starting address of the assembly language program has been specified by the programmer. The addresses in such a program are known as **absolute addresses**.

Assemblers

When translating assembly code into object code, the assembler must translate opcodes and format the bits in each instruction, and translate labels into addresses. In this section, we review the translation of assembly language into binary. Labels make the assembly process more complex, but they are the most important abstraction provided by the assembler. Labels let the programmer (a human programmer or a compiler generating assembly code) avoid worrying about the locations of instructions and data. Label processing requires making two passes through the assembly source code as follows:

1. The first pass scans the code to determine the address of each label.
2. The second pass assembles the instructions using the label values computed in the first pass.
As shown in Figure 2.17, the name of each symbol and its address is stored in a **symbol table** that is built during the first pass. The symbol table is built by scanning from the first instruction to the last.
During scanning, the current location in memory is kept in a **program location counter (PLC)**. Despite the similarity in name to a program counter, the PLC is not used to execute the program, only to assign memory locations to labels. For example, the PLC always makes exactly one pass through the program, whereas the program counter makes many passes over code in a loop. Thus, at the start of the first pass, the PLC is set to the program's starting address and the assembler looks at the first line. After examining the line, the assembler updates the PLC to the next location (since ARM instructions are four bytes long, the PLC would be incremented by four) and looks at the next instruction. If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The value of the label is equal to the current value of the PLC. At the end of the first pass, the assembler rewinds to the beginning of the assembly language file to make the second pass. During the second pass, when a label name is found, the label is looked up in the symbol table and its value substituted into the appropriate place in the



instruction.

But how do we know the starting value of the PLC? The simplest case is absolute addressing. In this case, one of the first statements in the assembly language program is a pseudo-op that specifies the **origin** of the program, that is, the location of the first address in the program. A common name for this pseudo-op (e.g., the one used for the ARM) is the ORG statement.

ORG 2000

Which puts the start of the program at location 2000. This pseudo-op accomplishes this by setting the PLC's value to its argument's value, 2000 in this case. Assemblers generally allow a program to have many ORG statements in case instructions or data must be spread around various spots in memory.

Linking:

Many assembly language programs are written as several smaller pieces rather than as a single large file. Breaking a large program into smaller files helps delineate program modularity. If the program uses library routines, those will already be preassembled, and assembly language source code for the libraries may not be available for purchase.

A **linker** allows a program to be stitched together out of several smaller pieces. The linker operates on the object files created by the assembler and modifies the assembled code to make the necessary links between files.

Some labels will be both defined and used in the same file. Other labels will be defined in a single file but used elsewhere as illustrated in Figure 2.18. The place in the file where a label is defined is known as an **entry point**. The place in the file where the label is used is called an **external reference**.

The main job of the loader is to resolve external references based on available entry points. As a result of the need to know how definitions and references connect, the assembler passes to the linker not only the object file but also the symbol table.

Even if the entire symbol table is not kept for later debugging purposes, it must at least pass the entry points. External references are identified in the object code by their relative symbol identifiers.

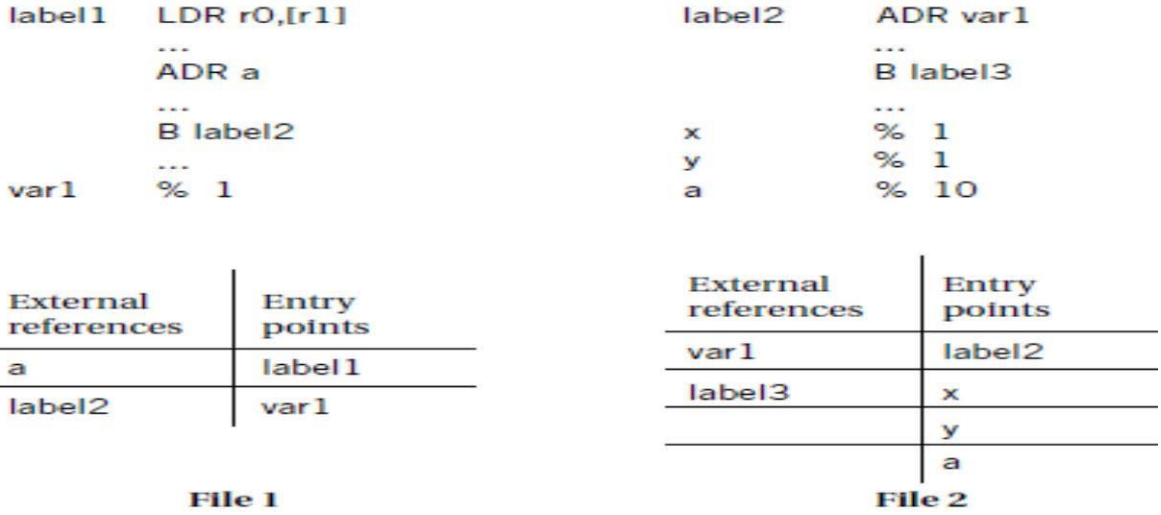


Fig. External references and entry points

The linker proceeds in two phases.

First, it determines the address of the start of each object file. The order in which object files are to be loaded is given by the user, either by specifying parameters when the loader is run or by creating a **load map** file that gives the order in which files are to be placed in memory. Given the order in which files are to be placed in memory and the length of each object file, it is easy to compute the starting address of each file.

At the start of the second phase, the loader merges all symbol tables from the object files into a single, large table. It then edits the object files to change relative addresses into addresses. This is typically performed by having the assembler write extra bits into the object file to identify the instructions and fields that refer to labels. If a label cannot be found in the merged symbol table, it is undefined and an error message is sent to the user.

11. Brief about basic compilation techniques. (6) [CO2-L1]

It is useful to understand how a high-level language program is translated into instructions. Since implementing an embedded computing system often requires controlling the instruction sequences used to handle interrupts, placement of data and

instructions in memory, and so forth, understanding how the compiler works can help you know when you cannot rely on the compiler.
Next, because many applications are also performance sensitive, understanding how code is generated can help you meet your performance goals, either by writing high-level code that gets compiled into the instructions you want or by recognizing when you must write your own assembly code.
The compilation process is summarized in Figure 2.19. Compilation begins with high-level language code such as C and generally produces assembly code. (Directly producing object code simply duplicates the functions of an assembler which is a very desirable stand-alone program to have.)
The high-level language program is parsed to break it into statements and expressions. In addition, a symbol table is generated, which includes all the named objects in the program. Some compilers may then perform higher-level optimizations that can be viewed as modifying the high-level language program input without reference to instructions.

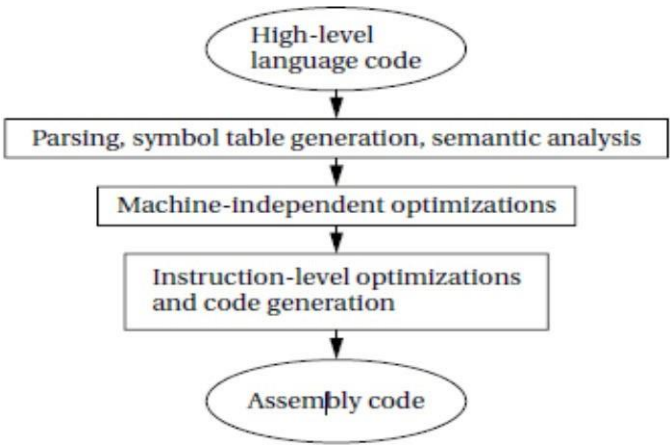
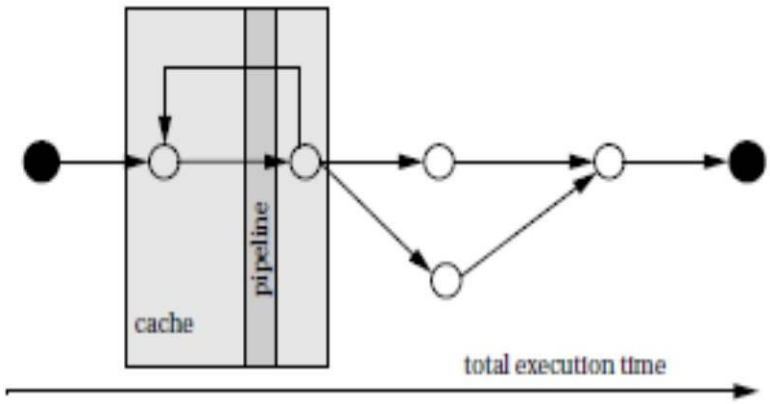


Fig.The compilation process

Simplifying arithmetic expressions is one example of a machine-independent optimization. Not all compilers do such optimizations, and compilers can vary widely regarding which combinations of machine-independent optimizations they do perform.
Instruction-level optimizations are aimed at generating code. They may work directly on real instructions or on a pseudo-instruction format that is later mapped onto the instructions of the target CPU. This level of optimization also helps modularize the compiler by allowing code generation to create simpler code that is later optimized. For example, consider the following array access code:
x[i] = c*x[i];
A simple code generator would generate the address for x[i] twice, once for each appearance in the statement. The later optimization phases can recognize this as an example of common expressions that need not be duplicated. While in this simple case it would be possible to create a code generator that never generated the redundant expression, taking into account every such optimization at code generation time is very difficult. We get better code and more reliable compilers by generating simple code first and then optimizing it.

12. Explain the methods used to analyze the program-level performance. (16) [CO2-L1]

Because embedded systems must perform functions in real time we often need to know how fast a program runs. The techniques we use to analyze program execution time are also helpful in analyzing properties such as power consumption. In this section, we study how to analyze programs to estimate their run times. We also examine how to optimize programs to improve their execution times; of course, optimization relies on analysis. It is important to keep in mind that CPU performance is not judged in the same way as program performance. Certainly, CPU clock rate is a very unreliable metric for program performance. But more importantly, the fact that the CPU executes part of our program quickly does not mean that it will execute the entire program at the rate we desire. As illustrated in Figure 5.22, the CPU pipeline and cache act as windows into our program. In order to understand the total execution time of our program, we must look at execution paths, which in general are far longer than the pipeline and cache windows. The pipeline and cache influence execution time ,but execution time is a global property of the program. While we might hope that the execution time of programs could be precisely determined, this is in fact difficult to do in practice:



- The execution time of a program often varies with the input data values because those values select different execution paths in the program. For example, loops may be executed a varying number of times, and different branches may execute blocks of varying complexity.
- The cache has a major effect on program performance, and once again, the cache's behavior depends in part on the data values input to the program.
- Execution times may vary even at the instruction level. Floating-point operations are the most sensitive to data values, but the normal integer execution pipeline can also introduce data-dependent variations. In general, the execution time of an instruction in a pipeline depends not only on that instruction but on the instructions around it in the pipeline.

We can measure program performance in several ways:

- Some microprocessor manufacturers supply simulators for their CPUs: The simulator runs on a workstation or PC, takes as input an executable for the microprocessor along with input data, and simulates the execution of that program. Some of these simulators

go beyond functional simulation to measure the execution time of the program. Simulation is clearly slower than executing the program on the actual microprocessor, but it also provides much greater visibility during execution. Be careful—some microprocessor performance simulators are not 100% accurate, and simulation of I/O-intensive code may be difficult.

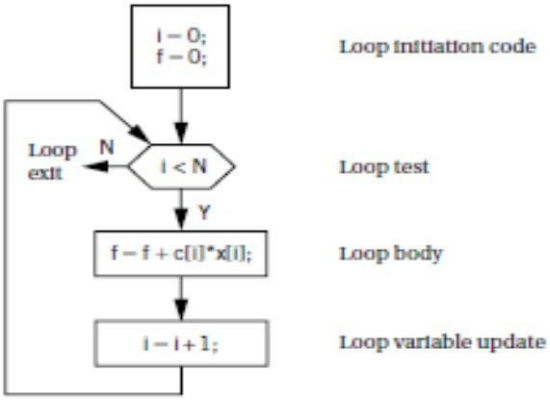
- A timer connected to the microprocessor bus can be used to measure performance of executing sections of code. The code to be measured would reset and start the timer at its start and stop the timer at the end of execution. The length of the program that can be measured is limited by the accuracy of the timer.
- A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment. This technique relies on the code being able to produce identifiable events on the bus to identify the start and stop of execution. The length of code that can be measured is limited by the size of the logic analyzers buffer. We are interested in the following three different types of performance measures on programs:
- **Average-case execution time** This is the typical execution time we would expect for typical data. Clearly, the first challenge is defining typical inputs.
- **Worst-case execution time** The longest time that the program can spend on any input sequence is clearly important for systems that must meet deadlines. In some cases, the input set that causes the worst-case execution time is obvious, but in many cases it is not.
- **Best-case execution time** This measure can be important in Multirate real-time systems

First, we look at the fundamentals of program performance in more detail. We then consider trace-driven performance based on executing the program and observing its behavior.

Elements of Program Performance:

The path is the sequence of instructions executed by the program (or its equivalent in the high-level language representation of the program). The instruction timing is determined based on the sequence of instructions traced by the program path, which takes into account data dependencies, pipeline behaviour, and caching. Luckily, these two problems can be solved relatively independently.

Although we can trace the execution path of a program through its high-level language specification, it is hard to get accurate estimates of total execution time from a high-level language program. The number of memory locations and variables must be estimated, and results may be either saved for reuse or recomputed on the fly, among other effects. These problems become more challenging as the compiler puts more and more effort into optimizing the program. However, some aspects of program performance can be estimated by looking directly at the C program. For example, if a program contains a loop with a large, fixed iteration bound or if one branch of a conditional is much longer than another, we can get at least a rough idea that these are more time-consuming.



Segments of the program. Of course ,a precise estimate of performance also relies on the instructions to be executed, since different instructions take different amounts of time. (In addition, to make life even more difficult, the execution time of one instruction can depend on the instructions executed before and after it

To measure the longest path length ,we must find the longest path through the optimized CDFG since the compiler may change the structure of the control and data flow to optimize the program's implementation. It is important to keep in mind that choosing the longest path through a CDFG as measured by the number of nodes or edges touched may not correspond to the longest execution time. Since the execution time of a node in the CDFG will vary greatly depending on the instructions represented by that node, we must keep in mind that the longest path through the CDFG depends on the execution times of the nodes. In general, it is good policy to choose several of what we estimate are the longest paths through the program and measure the lengths of all of them in sufficient detail to be sure that we have in fact captured the longest path. Once we know the execution path of the program, we have to measure the execution time of the instructions executed along that path. The simplest estimate is to assume that every instruction takes the same number of clock cycles, whichmeans we need only count the instructions and multiply by the per-instruction execution time to obtain n the program's total execution time. However, even ignoring cache effects, this technique is simplistic for the reasons summarized below.

- Not all instructions take the same amount of time. Although RISC architectures tend to provide uniform instruction execution times in order to keep the CPU's pipeline full, even many RISC architectures take different amounts of time to execute certain instructions. Multiple load-store instructions are examples of longer-executing instructions in the ARM architecture. Floating point
instructions show especially wide variations in execution time—while basic multiply and add operations are fast, some transcendental functions can take thousands of cycles to execute.
- Execution times of instructions are not independent. The execution time of one instruction depends on the instructions around it. For example ,many CPUs use register bypassing to speed up instruction sequences when the result of one instruction is used in the next instruction. As a result, the execution
time of an instruction may depend on whether its destination register is used as a source for the next operation (or vice versa).

▪ The execution time of an instruction may depend on operand values. This is clearly true of floating-point instructions in which a different number of iterations may be required to calculate the result. Other specialized instructions can, for example, perform a data-dependent number of integer operations.

Measurement-Driven Performance Analysis:

Most methods of measuring program performance combine the determination of the execution path and the timing of that path: as the program executes, it chooses a path and we observe the execution time along that path. We refer to the record of the execution path of a program as a **program trace** (or more succinctly ,a **trace**). Traces can be valuable for other purposes, such as analyzing the cache behavior of the program. Perhaps the biggest problem in measuring program performance is figuring out a useful set of inputs to provide to the program. This problem has two aspects. First, we have to determine the actual input values. We may be able to use benchmark data sets or data captured from a running system to help us generate typical values. For simple programs, we may be able to analyze the algorithm to determine the inputs that cause the worst-case execution time.

The other problem with input data is the **software scaffolding** that we may need to feed data into the program and get data out. When we are designing a large system, it may be difficult to extract out part of the software and test it independently of the other parts of the system. We may need to add new testing modules to the system software to help us introduce testing values and to observe testing outputs.

We can measure program performance either directly on the hardware or by using a simulator. Each method has its advantages and disadvantages. Physical measurement requires some sort of hardware instrumentation. The most direct method of measuring the performance of a program would be to watch the program counter's value: start a timer when the PC reaches the program's start, stop the timer when it reaches the program's end. Unfortunately, it generally isn't possible to directly observe the program counter. However, it is possible in many cases to modify the program so that it starts a timer at the beginning of execution and stops the timer at the end. While this doesn't give us direct information about the program trace, it does give us execution time. If we have several timers available, we can use them to measure the execution time of different parts of the program.

A logic analyzer or an oscilloscope can be used to watch for signals that mark various points in the execution of the program. However, because logic analyzers have a limited amount of memory, this approach doesn't work well for programs with extremely long execution times. Some CPUs have hardware facilities for automatically generating trace information. For example, the Pentium family microprocessors generate a special bus cycle, a branch trace message, that shows the source and/or destination address of a branch [Col97]. If we record only traces, we can reconstruct the instructions executed within the basic blocks while greatly reducing the amount of memory required to hold the trace. The alternative to physical measurement of execution time is simulation. A CPU simulator is a program that takes as input a memory image for a CPU and performs the operations on that memory image that the actual CPU would perform, leaving the results in the modified memory image.

For purposes of performance analysis, the most important type of CPU simulator is the **cycle-accurate simulator**, which performs a sufficiently detailed simulation of the processor's internals so that it can determine the exact number of clock cycles required

for execution. A cycle-accurate simulator is built with detailed knowledge of how the processor works, so that it can take into account all the possible behaviours of the micro architecture that may affect execution time. Cycle-accurate simulators are slower than the processor itself, but a variety of techniques can be used to make them surprisingly fast, running only hundreds of times slower than the hardware itself. A cycle-accurate simulator has a complete model of the processor, including the cache. It can therefore provide valuable information about why the program runs too slowly. The next example discusses a simulator that can be used to model many different processors.

13. What is optimization? Also explain software performance optimization. (6) [CO2-L1]

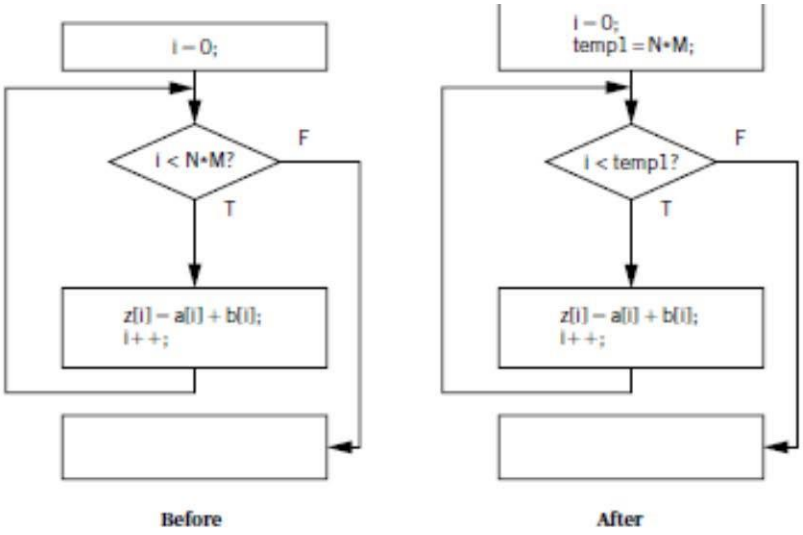
Loop Optimizations:

Loops are important targets for optimization because programs with loops tend to spend a lot of time executing those loops. There are three important techniques in optimizing loops: **code motion, induction variable elimination,** and **strength reduction**. Code motion lets us move unnecessary code out of a loop. If a computation's result does not depend on operations performed in the loop body, then we can safely move it out of the loop. Code motion opportunities can arise because programmers may find some computations clearer and more concise when put in the loop body, even though they are not strictly dependent on the loop iterations. A simple example of code motion is also common

```
for (i = 0; i < N*M; i++) {
     z[i] = a[i] + b[i];
     }
```

The code motion opportunity becomes more obvious when we draw the loop's CDFG as shown in Figure 5.23.The loop bound computation is performed on every iteration during the loop test, even though the result never changes. We can avoid N *_M _*1 unnecessary executions of this statement by moving it before the loop, as shown in the figure. An **induction variable** is a variable whose value is derived from the loop iteration variable's value. The compiler often introduces induction variables to help it implement the loop. Properly transformed, we may be able to eliminate some variables and apply strength reduction to others. A nested loop is a good example of the use of induction variables. Here is a simple nested loop

```
for (i = 0; i < N; i++)
     for (j = 0; j < M; j++)
          z[i][j] = b[i][j];
```



Before After

Cache Optimizations

A **loop nest** is a set of loops, one inside the other. Loop nests occur when we process arrays. A large body of techniques has been developed for optimizing loop nests. Rewriting a loop nest changes the order in which array elements are accessed. This can expose new parallelism opportunities that can be exploited by later stages of the compiler, and it can also improve cache performance

14. Explain the following. Analysis and optimization of execution time, power, energy, program size. (6) [CO2-L1]

- The memory footprint of a program is determined by the size of its data and instructions. Both must be considered to minimize program size.

- Data provide an excellent opportunity for minimizing size because the data are most highly dependent on programming style. Because inefficient programs often keep several copies of data, identifying and eliminating duplications can lead to significant memory savings usually with little performance penalty.

- Buffers should be sized carefully rather than defining a data array to a large size that the program will never attain, determine the actual maximum amount of data held in the buffer and allocate the array accordingly. Data can sometimes be packed, such as by storing several flags in a single word and extracting them by using bit-level operations.
- A very low-level technique for minimizing data is to reuse values. For instance, if several constants happen to have the same value, they can be mapped to the same location. Data buffers can often be reused at several different points in the program. This technique must be used with extreme caution, however, since subsequent versions of the program may not use the same values for the constants.

- A more generally applicable technique is to generate data on the fly rather than store it. Of course, the code required to generate the data takes up space in the

program, but when complex data structures are involved there may be some net space savings from using code to generate data.

- Minimizing the size of the instruction text of a program requires a mix of high-level program transformations and careful instruction selection.
- Encapsulating functions in subroutines can reduce program size when done carefully. Because subroutines have overhead for parameter passing that is not obvious from the high-level language code, there is a minimum-size function body for which a subroutine makes sense.
- Architectures that have variable-size instruction lengths are particularly good candidates for careful coding to minimize program size, which may require assembly language coding of key program segments. There may also be cases in which one or a sequence of instructions is much smaller than alternative implementations for example, a multiply-accumulate instruction may be both smaller and faster than separate arithmetic operations.
- When reducing the number of instructions in a program, one important technique is the proper use of subroutines. If the program performs identical operations repeatedly, these operations are natural candidates for subroutines.
- Even if the operations vary somewhat, you may be able to construct a properly parameterized subroutine that saves space. Of course, when considering the code size savings, the subroutine

15. Discus in detail about program validation and testing. (16) [CO2-H1]

Complex systems need testing to ensure that they work as they are intended. But bugs can be subtle, particularly in embedded systems, where specialized hardware and real-time responsiveness make programming more challenging. Fortunately, there are many available techniques for software testing that can help us generate a comprehensive set of tests to ensure that our system works properly.

. In this section, we concentrate on nuts-and-bolts techniques for creating a good set of tests for a given program. The first question we must ask ourselves is how much testing is enough. Clearly, we cannot test the program for every possible combination of inputs. Because we cannot implement an infinite number of tests, we naturally ask ourselves what a reasonable standard of thoroughness is. One of the major contributions of software testing is to provide us with standards of thoroughness that make sense. Following these standards does not guarantee that we will find all bugs. But by breaking the testing problem into sub problems and analyzing each sub problem.

we can identify testing methods that provide reasonable amounts of testing while keeping the testing time within reasonable bounds. The two major types of testing strategies:

- **Black-box** methods generate tests without looking at the internal structure of the program.
- **Clear-box** (also known as **white-box**) methods generate tests based on the program structure.

In this section we cover both types of tests, which complement each other by exercising programs in very different ways

Clear-Box Testing

The control/data flowgraph extracted from a program's source code is an important tool in developing clear-box tests for the program. To adequately test the program, we must

exercise both its control and data operations. In order to execute and evaluate these tests ,we must be able to control variables in the program and observe the results of computations, much as in manufacturing testing. In general,we may need to modify the program to make it more testable. By adding new inputs and outputs, we can usually substantially reduce the effort required to find and execute the test. Example 5.11 illustrates the importance of observability and controllability in software testing. No matter what we are testing,we must accomplish the following three things in a test:

- Provide the program with inputs that exercise the test we are interested in.
- Execute the program to perform the test.
- Examine the outputs to determine whether the test was successful..

Being able to perform this process for a large number of tests entails some amount of drudgery, but that drudgery can be alleviated with good program design that simplifies controllability and observability. The next task is to determine the set of tests to be performed.We need to perform
many different types of tests to be confident that we have identified a large fraction of the existing bugs.

Black-Box Testing

Black-box tests are generated without knowledge of the code being tested. When used alone black-box tests have a low probability of finding all the bugs in a program. But when used in conjunction with clear-box tests they help provide a well-rounded test set, since black-box tests are likely to uncover errors that are unlikely to be found by tests extracted from the code structure. Black-box tests can really work. For instance, when asked to test an instrument whose front panel was run by a microcontroller, one acquaintance of the author used his hand to depress all the buttons simultaneously. The front panel immediately locked up. This situation could occur in practice if the instrument were placed face-down on a table, but discovery of this bug would be very unlikely via clear-box tests. One important technique is to take tests directly from the specification for the code under design.

The specification should state which outputs are expected for certain inputs. Tests should be created that provide specified outputs and evaluate whether the results also satisfy the inputs. We can’t test every possible input combination, but some rules of thumb help us select reasonable sets of inputs. When an input can range across a set of values, it is a very good idea to test at the ends of the range. For example, if an input must be between 1 and 10, 0, 1, 10, and 11 are all important values to test. We should be sure to consider tests both within and outside the range, such as, testing values within the range and outside the range. We may want to consider tests well outside the valid range as well as boundary-condition tests.

Random tests form one category of black-box test. Random values are generated with a given distribution. The expected values are computed independently of the system, and then the test inputs are applied. A large number of tests must be applied for the results to be statistically significant, but the tests are easy to generate. Another scenario is to test certain types of data values. For example, integer valued inputs can be generated at interesting values such as 0,1,and values near the
maximum end of the data range. Illegal values can be tested as well.

Regression tests form an extremely important category of tests. When tests are created during earlier stages in the system design or for previous versions of the system, those tests should be saved to apply to the later versions of the system.

Clearly, unless the system specification changed, the new system should be able to pass old tests. In some cases old bugs can creep back into systems, such as when an old version of a software module is inadvertently installed. In other cases regression tests simply exercise the code in different ways than would be done for the current version of the code and therefore possibly exercise different bugs. Some embedded systems, particularly digital signal processing systems, lend themselves to numerical analysis. Signal processing algorithms are frequently implemented with limited-range arithmetic to save hardware costs. Aggressive data sets can be generated to stress the numerical accuracy of the system. These tests can often be generated from the original formulas without reference to the source code

Evaluating Function Tests

How much testing is enough? Horgan and Mathur [Hor96] evaluated the coverage of two well-known programs, TeX and awk. They used functional tests for these programs that had been developed over several years of extensive testing. Upon applying those functional tests to the programs, they obtained the code coverage statistics shown in Figure 5.30.The columns refer to various types of test coverage: block refers to basic blocks, decision to conditionals, p-use to a use of a variable in a predicate (decision), and c-use to variable use in a non predicate computation. These results are at least suggestive that functional testing does not fully exercise the code and that techniques that explicitly generate tests for various pieces of code are necessary to obtain adequate levels of code coverage. Methodological techniques are important for understanding the quality of your tests. For example, if you keep track of the number of bugs tested each day, the data you collect over time should show you some trends on the number of errors per page of code to expect on the average, how many bugs are caught by certain kinds of tests, and so on. We address methodological approaches to quality control in more detail. One interesting method for analyzing the coverage of your tests is **error injection**.

First, take your existing code and add bugs to it, keeping track of where the bugs were added. Then run your existing tests on the modified program. By counting the number of added bugs your tests found, you can get an idea of how effective.

	Block	Decision	P-use	C-use
TeX	85%	72%	53%	48%
awk	70%	59%	48%	55%

the tests are in uncovering the bugs you haven't yet found. This method assumes that you can deliberately inject bugs that are of similar varieties to those created naturally by programming errors.

If the bugs are too easy or too difficult to find or simply require different types of tests, then bug injection's results will not be relevant. Of course, it is essential that you finally use the correct code, not the code with added bugs.

Unit - III

Processes and Operating Systems

Part – A

1. What are the major inter process communication mechanism? [CO3-L1-April 2014]

- ✓ Shared memory communication
- ✓ Message passing.

2. Define context switching. [CO3-L2-April 2014, Nov/Dec 2013 & Nov/Dec 2012]

- ✓ A context switch is the computing process of storing and restoring of a CPU so that execution can be resumed from the same point at a later time.
- ✓ The context switching is an essential feature of multitasking operation system.

3. Define: processes. [CO3-L2-Nov/Dec 2013]

- ✓ A process is a single execution of a program.
- ✓ If we run the same program two different times, we have created two different processes.
- ✓ Each process has its own state that includes not only its register but also all of its memory.

4. List the process of scheduling policies. [CO3-L1-May/June 2013]

- ✓ Cyclo static scheduling
- ✓ Time division multiple access scheduling
- ✓ Round robin scheduling.

5. Define queue. [CO3-L1]

In computer technology, a queue is a sequence of work objects that are waiting to be processed.

6. Define stack. [CO3-L1]

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

7. What is the use of interrupts service routines? [CO3-L1-May/June 2012]

- ✓ Input/output data transfer for peripheral devices.
- ✓ Input signals to be used for timing purpose.
- ✓ Real time executives/multitasking
- ✓ Event driven program.

8. What are the three conditions that must be satisfied by the re-entrant function? [CO3-L1-May/June 2012]

A function is called re-entrant function when the following three conditions are satisfied

- ✓ All the arguments pass the values and some of the argument is a pointer whenever a calling function calls it.
- ✓ When an operation is not atomic, the function should not operate on any variable, which is declared but passed by reference not passed by arguments in to the function.
- ✓ That function does not call any other function that is not itself re-entrant.

9. What is meant by operating system? [CO3-L1]

An operating system (OS) is software that manages computer hardware and software resources and provides common services for computer programs. The operating system is an essential component of the system software in a computer system. Application programs usually require an operating system to function.

10. What is termed as task? [CO3-L1]

A task is a set of computations or actions that processes on a CPU under the control of a scheduling kernel. It also has a process control structure called a task control block that saves at the memory. It has a unique ID. It has states in the system as follows: idle, ready, running, blocked and finished.

11. What are the states of a process? [CO3-L1]

Processes may be in any one of the 5 states,

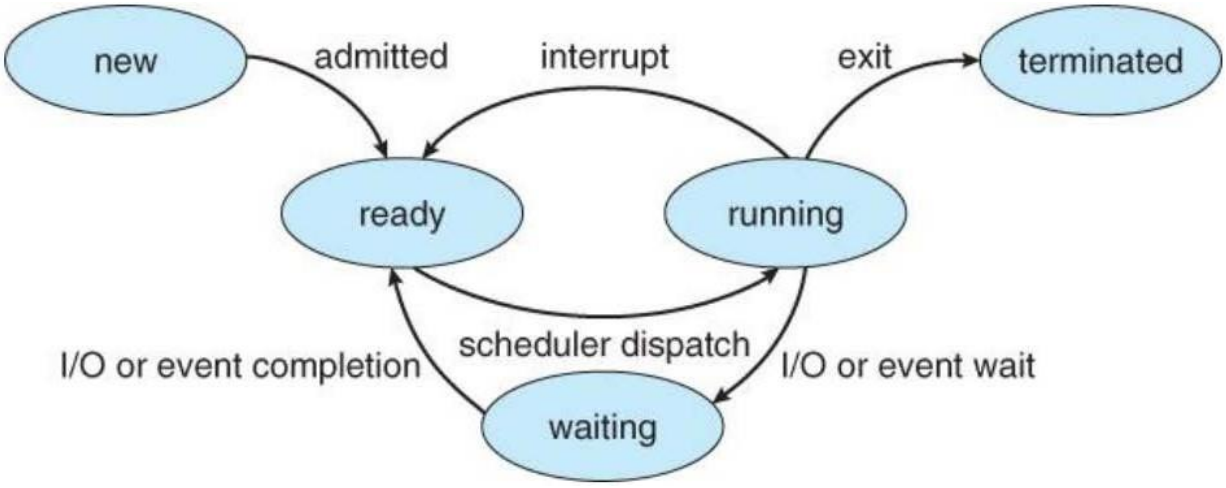
a. New - The process is in the stage of being created.

b. Ready - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.

c. Running - The CPU is working on this process's instructions.

d. Waiting - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.

e. Terminated - The process has completed.



12. Define Scheduling? [CO3-L1]

This is defined as a process of selection which says that a process has the right to use the processor at given time.

13. What is scheduling policy? [CO3-L1]

It says the way in which processes are chosen to get promotion from ready state to running state.

14. What is scheduling overhead? [CO3-L1]

It is defined as time of execution needed to select the next execution process.

15. Define priority scheduling? [CO3-L1]

A simple scheduler maintains a priority queue of processes that are in the runnable state.

16. Define earliest deadline first (EDF) scheduling? [CO3-L1]

Earliest deadline first (EDF) or least time to go is a dynamic scheduling algorithm used in real-time operating systems to place processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be earched for the process closest to its deadline. This process is the next to be scheduled for execution.EDF is an

optimal scheduling algorithm on preemptive uniprocessors, in the following sense: if a collection of independent jobs, each characterized by an arrival time, an execution requirement and a deadline, can be scheduled (by any algorithm) in a way that ensures all the jobs complete by their deadline, the EDF will schedule this collection of jobs so they all complete by their deadline.

17. What is the function in ready state? [CO3-L1]

The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions. Processes in the Ready state are placed in the ready queue.

18. What are the types of Scheduling? [CO3-L1]

- ✓ Long term scheduling: which determines which programs are admitted to the system for execution and when, and which ones should be exited.
- ✓ Medium term scheduling: which determines when processes are to be suspended and resumed;
- ✓ Short term scheduling (or dispatching): which determines which of the ready processes can have CPU resources, and for how long.

19. What is multirate system? [CO3-L1]

Multirate simply means "multiple sampling rates". A multirate DSP system uses multiple sampling rates within the system. Whenever a signal at one rate has to

be used by a system that expects a different rate, the rate has to be increased or decreased, and some processing is required to do so. Therefore "Multirate DSP" really refers to the art or science of changing sampling rates.

20. What is Multiprocessing, Multithreading? [CO3-L1]

Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them.Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer.

21. What is meant by RTOS? [CO3-L1]

An RTOS is an OS for response time controlled and event controlled processes. RTOS is an OS for embedded systems, as these have real time programming issues to solve.

22. Define RMS (Rate Monotonic Scheduling)? [CO3-L1]

In computer science, rate-monotonic scheduling (RMS) is a scheduling algorithm used in real-time operating systems with a static-priority scheduling class. The static priorities are assigned on the basis of the cycle duration of the job: the shorter the cycle duration is, the higher is the job's priority.

23. What are the advantages & disadvantages of Assembly language? [CO3-L1]

Advantages

- ✓ The symbolic programming of Assembly Language is easier to understand and saves a lot of time and effort of the programmer.
- ✓ It is easier to correct errors and modify program instructions.
- ✓ Assembly Language has the same efficiency of execution as the machine level language. Because this is one-to-one translator between assembly language program and its corresponding machine language program.

Disadvantages

- ✓ One of the major disadvantages is that assembly language is machine dependent. A program written for one computer might not run in other computers with different hardware configuration.

24. What are advantages of high level languages? [CO3-L1]

- ✓ Easy to Learn
- ✓ Easy to Understand
- ✓ Easy to Write Program
- ✓ Easy to Detect & Remove Errors
- ✓ Built-in Library Functions
- ✓ Machine Independence

25. What are the goals of RTOS? [CO3-L1]

- ✓ Facilitating easy sharing of resources,
- ✓ Facilitating easy implantation of the application software,
- ✓ Maximizing system performance,
- ✓ Providing management functions for the processes, memory, and I/Os and for other functions for which it is designed.
- ✓ Providing management and organization functions for the devices and files and file like devices.
- ✓ Portability
- ✓ Interoperability
- ✓ Providing common set of interfaces

Part - B

1. What are Tasks and Processes? And also explain tasks and processes. (8) [CO3-L2]

Tasks and Processes

- Many (if not most) embedded computing systems do more than one thing that is, the environment can cause mode changes that in turn cause the embedded system to behave quite differently. For example, when designing a telephone answering machine,

- We can define recording a phone call and operating the user's control panel as distinct tasks, because they perform logically distinct operations and they must be performed at very different rates. These different **tasks** are part of the system's functionality, but that application-level organization of functionality is often reflected in the structure of the program as well.

- A **process** is a single execution of a program. If we run the same program two different times, we have created two different processes. Each process has its own state that includes not only its registers but all of its memory. In some OSs, the memory management unit is used to keep each process in a separate address space. In others, particularly lightweight RTOSs, the processes run in the same address space. Processes that share the same address space are often called **threads**.
- As shown in Figure, this device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line, based on a predefined compression table. Such a box may be used, for example, to compress data being sent to a modem.
- The program's need to receive and send data at different rates for example, the program may emit 2 bits for the first byte and then 7 bits for the second byte will obviously find itself reflected in the structure of the code. It is easy to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of output bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets.

- But beyond the need to create a clean data structure that simplifies the control structure of the code, we must also ensure that we process the inputs and outputs at the proper rates. For example, if we spend too much time in packaging and emitting output characters, we may drop an input character. Solving timing problems is a more challenging problem.
- The text compression box provides a simple example of rate control problems. A control panel on a machine provides an example of a different type of rate control problem, the **asynchronous input**.
- The control panel of the compression box may, for example, include a compression mode button that disables or enables compression, so that the input text is passed through unchanged when compression is disabled. We certainly do not know when the user will push the compression mode button the button may be depressed asynchronously relative to the arrival of characters for compression.

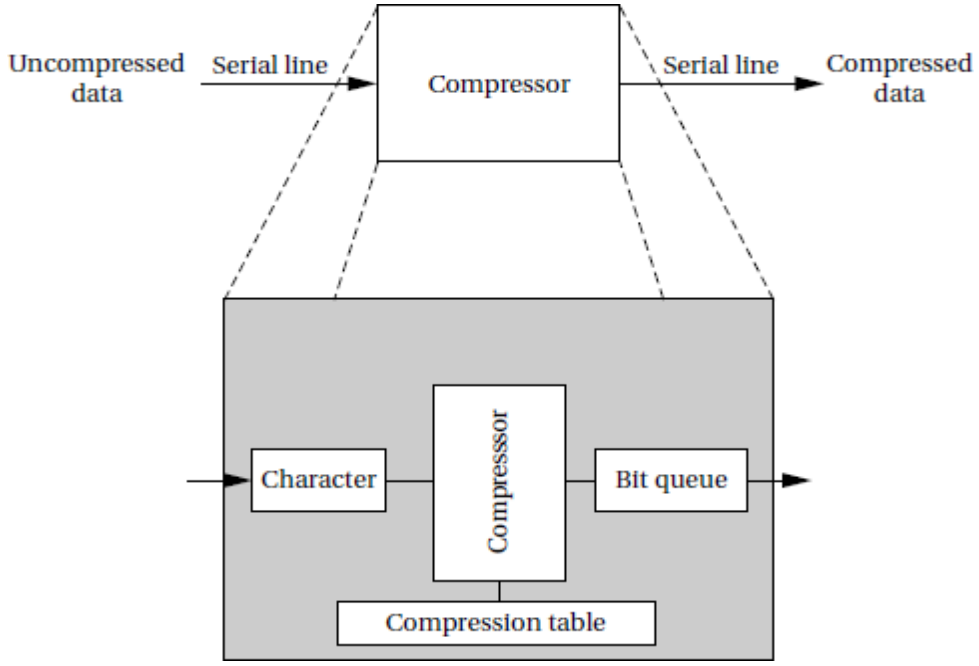


Fig An on-the-fly compression box.

2. Define and describe Multirate Systems. (10) [CO3-L1]

Implementing code that satisfies timing requirements is even more complex when multiple rates of computation must be handled. **Multirate** embedded computing systems are very common, including automobile engines, printers, and cell phones. In all these systems, certain operations must be executed periodically, and each operation is executed at its own rate.

Timing Requirements on Processes

- Processes can have several different types of timing requirements imposed on them by the application. The timing requirements on a set of processes strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid. Before studying scheduling proper, we

outline the types of process timing requirements that are useful in embedded system design.

- Figure 3.2 illustrates different ways in which we can define two important requirements on processes: **release time** and **deadline**.

- The release time is the time at which the process becomes ready to execute; this is not necessarily the time at which it actually takes control of the CPU and starts to run. An aperiodic process is by definition initiated by an event, such as external data arriving or data computed by another process.
- The release time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself. For a periodically executed process, there are two common possibilities.
- In simpler systems, the process may become ready at the beginning of the period. More sophisticated systems, such as those with data dependencies between processes, may set the release time at the arrival time of certain data, at a time after the start of the eriod.

- A deadline specifies when a computation must be finished. The deadline for an aperiodic process is generally measured from the release time, since that is the only reasonable time reference. The deadline for a periodic process may in general occur at some time other than the end of the period.

- Rate requirements are also fairly common. A rate requirement specifies how quickly processes must be initiated.

- The **period** of a process is the time between successive executions. For example, the period of a digital filter is defined by the time interval between successive input samples.

- The process's **rate** is the inverse of its period. In a multirate system, each process executes at its own distinct rate.

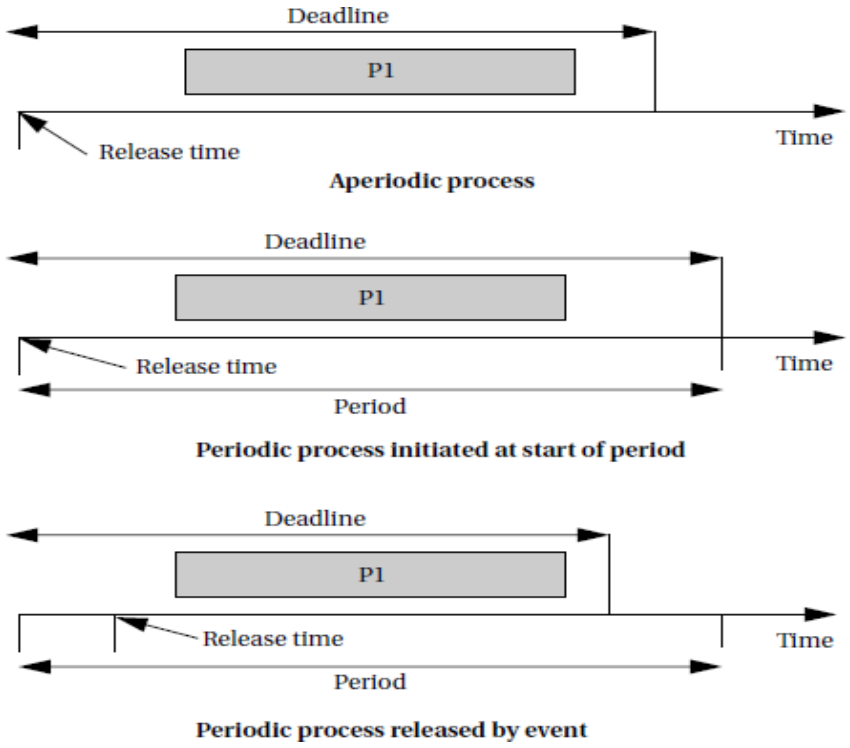


Fig 3.2 Example definitions of release times and deadlines.

- The most common case for periodic processes is for the initiation interval to be equal to the period. However, pipelined execution of processes allows the initiation interval to be less than the period. Figure 3.3 illustrates process execution in a system with four CPUs.

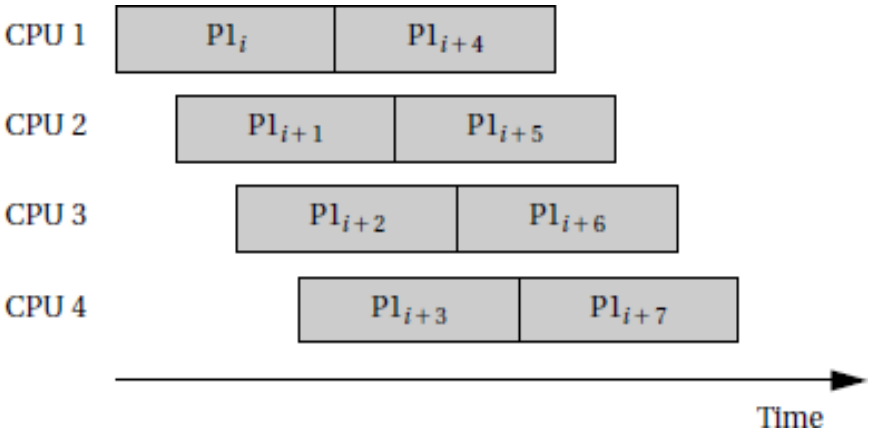


Fig 3.3 A sequence of processes with a high initiation rate.

CPU Metrics

We also need some terminology to describe how the process actually executes. The **initiation time** is the time at which a process actually starts

executing on the CPU. The **completion time** is the time at which the process finishes its work.

The most basic measure of work is the amount of **CPU time** expended by a process. The CPU time of process i is called Ci . Note that the CPU time is not equal to the completion time minus initiation time; several other processes may interrupt execution. The total CPU time consumed by a set of processes is

$$T = \sum T_i \quad (3.1)$$

We need a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is **utilization**:

U=CPU time for useful work/total available CPU time

(3.2) Utilization is the ratio of the CPU time that is being used for useful computations to the total

available CPU time. This ratio ranges between 0 and 1, with 1 meaning that all of the available CPU time is being used for system purposes. The utilization is often expressed as a percentage. If we measure the total execution time of all processes over an interval of time t, then the CPU utilization is

$$U=T/t. \quad (3.3)$$

3. Explain about Operating systems and context switching. [CO3-L1]

The best way to understand processes and context is to dive into an RTOS implementation. We will use the FreeRTOS.org kernel as an example; in particular, we will use version 4.7.0 for the ARM7

AT91 platform. A process is known in FreeRTOS.org as a task. Task priorities in FreeRTOS.org are ranked opposite to the convention we use in the rest of the book: higher numbers denote higher priorities and the priority 0 task is the idle task.

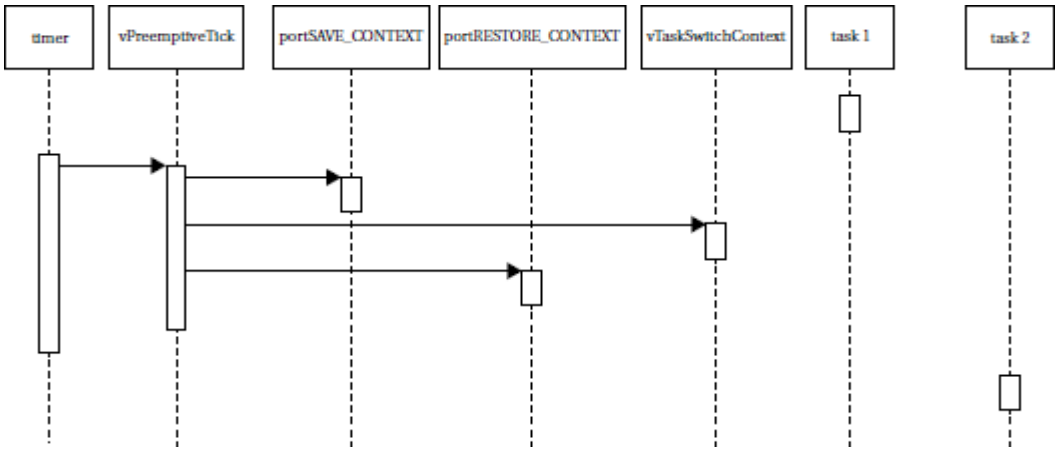


Fig 3.4 Sequence diagram for freeRTOS.org context switch.

To understand the basics of a context switch, let's assume that the set of tasks is in steady state: Everything has been initialized, the OS is running, and we are ready for a timer interrupt. Figure 3.4 shows a sequence diagram for a context switch in freeRTOS.org. This diagram shows the application tasks, the hardware timer, and all the functions in the kernel that are involved in the context switch:

- vPreemptiveTick () is called when the timer ticks.
- portSAVE_CONTEXT() swaps out the current task context..
- vTaskSwitchContext ( ) chooses a new task.
- portRESTORE_CONTEXT() swaps in the new context.

☐ An Operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.

☐ A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being applications programs.

☐ An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

4. Discus the various schduling policies in realtime operating system. (8) [CO3-L1]

☐ A **scheduling policy** defines how processes are selected for promotion from the ready state to the running state. Every multitasking OS implements some type of scheduling policy. Choosing the right scheduling policy not only ensures that the system will meet all its timing requirements, but it also has a profound influence on the CPU horsepower required to implement the system's functionality.

☐ Schedulability means whether there exists a schedule of execution for the processes in a system that satisfies all their timing requirements. In general, we must construct a schedule to show schedulability, but in some cases we can eliminate some sets of processes as unschedulable using some very simple tests.

☐ Utilization is one of the key metrics in evaluating a scheduling policy. Our most basic requirement is
that CPU utilization be no more than 100% since we can't use the CPU more than 100% of the time.

☐ When we evaluate the utilization of the CPU, we generally do so over a finite period that covers all possible combinations of process executions. For periodic

processes, the length of time that must be considered is the **hyperperiod**, which is the least-common multiple of the periods of all the processes. (The complete schedule for the least-common multiple of the periods is sometimes called the **unrolled schedule**.) If we evaluate the hyperperiod, we are sure to have considered all possible combinations of the periodic processes.

☐ We will see that some types of timing requirements for a set of processes imply that we cannot utilize 100% of the CPU's execution time on useful work, even ignoring context switching overhead. However, some scheduling policies can deliver higher CPU utilizations than others, even for the same timing requirements.

☐ One very simple scheduling policy is known as **cyclostatic** scheduling or sometimes as **Time Division Multiple Access** scheduling. As illustrated in Figure 3.5, a cyclostatic schedule is divided into equal-sized time slots over an interval equal to the length of the hyperperiod H. Processes always run in the same time slot.

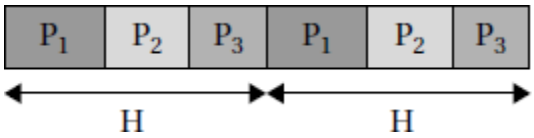


Fig 3.5 Cyclostatic scheduling.

- Two factors affect utilization: the number of time slots used and the fraction of each time slot that is used for useful work. Depending on the deadlines for some of the processes, we may need to leave some time slots empty. And since the time slots are of equal size, some short processes may have time left over in their time slot

- Another scheduling policy that is slightly more sophisticated is **round robin**. As illustrated in Figure 3.6, round robin uses the same hyperperiod as does cyclostatic. It also evaluates the processes in order.

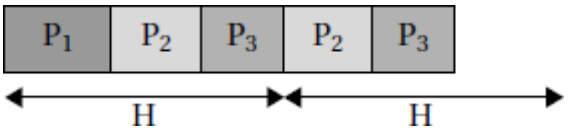


Fig 3.6 Round-robin scheduling.

- But unlike cyclostatic scheduling, if a process does not have any useful work

to do, the round- robin scheduler moves on to the next process in order to fill the time slot with useful work. In this example, all three processes execute during the first hyperperiod, but during the second one, P1 has no useful work and is skipped.

- The processes are always evaluated in the same order. The last time slot in the hyperperiod is left empty; if we have occasional, non-periodic tasks without deadlines, we can execute them in these empty time slots. Round-robin scheduling is often used in hardware such as buses because it is very simple to implement but it provides some amount of flexibility.

- In addition to utilization, we must also consider **scheduling overhead**—the execution time required to choose the next execution process, which is incurred in addition to any context switching overhead.

- In general, the more sophisticated the scheduling policy, the more CPU time it takes during system operation to implement it. Moreover, we generally achieve higher theoretical CPU utilization by applying more complex scheduling policies with higher overheads.

- The final decision on a scheduling policy must take into account both theoretical utilization and practical scheduling overhead.

5. Write short notes on inter process communication mechanisms. (16) [CO3-L1]

☐ Processes often need to communicate with each other. **Interprocess communication mechanisms** are provided by the operating system as part of the process abstraction.

☐ In general, a process can send a communication in one of two ways: **blocking** or **onblocking**.
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**.

Shared Memory Communication:

☐ Figure 3.9 illustrates how shared memory communication works in a bus-based system. Two components, such as a CPU and an I/O device, communicate through a shared memory location. The software on the CPU has been designed to know the address of the shared location.

☐ The shared location has also been loaded into the proper register of the I/O device. If, as in the figure, the CPU wants to send data to the device, it writes

to the shared location. The I/O device then reads the data from that location. The read and write operations are standard and can be encapsulated in a procedural interface.

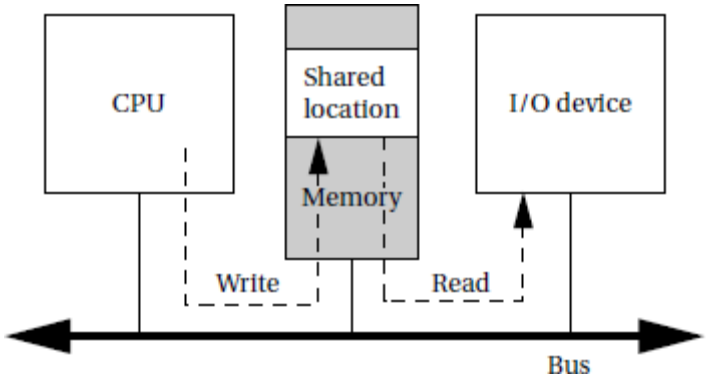


Fig 3.9 Shared memory communication implemented on a bus.

☐As an application of shared memory, let us consider the situation of Figure 6.14in which the CPU and the I/O device want to communicate through a shared memory block. There must be a flag that tells the CPU when the data from the I/O device is ready.

☐The flag, an additional shared data location, has a value of 0 when the data are not ready and 1 when the data are ready. If the flag is used only by the CPU, then the flag can be implemented using a standard memory write operation. If the same flag is used for bidirectional signaling between the CPU and the I/O device, care must be taken. Consider the following scenario:

1. CPU reads the flag location and sees that it is 0.

2. I/O device reads the flag location and sees that it is 0.

3. CPU sets the flag location to 1 and writes data to the shared location.

4. I/O device erroneously sets the flag to 1 and overwrites the data left by the CPU.

Message Passing:

- Message passing communication complements the shared memory model. As shown in Figure, 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.

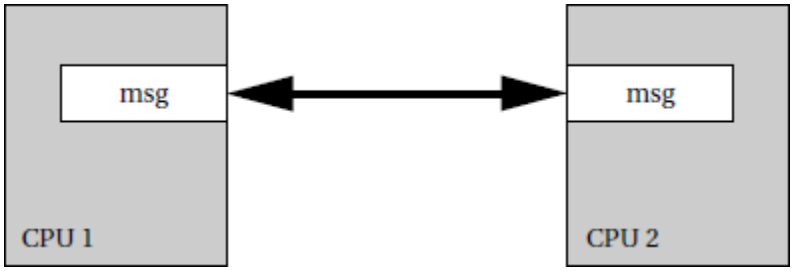


Fig Message passing communication.

- Applications in which units operate relatively autonomously are natural candidates for message passing communication. For example, a home control system has one microcontroller per household device—lamp, thermostat, faucet, appliance, and so on.

- The devices must communicate relatively infrequently; furthermore, their physical separation is large enough that we would not naturally think of them as sharing a central pool of memory.

- Passing communication packets among the devices is a natural way to describe coordination between these devices. Message passing is the natural implementation of communication in many 8-bit microcontrollers that do not normally operate with external memory.

Signals

Another form of interprocess communication commonly used in Unix is the **signal**. A signal is simple because it does not pass data beyond the existence of the signal itself. A signal is analogous to an interrupt, but it is entirely a software creation. A signal is generated by a process and transmitted to another process by the operating system.

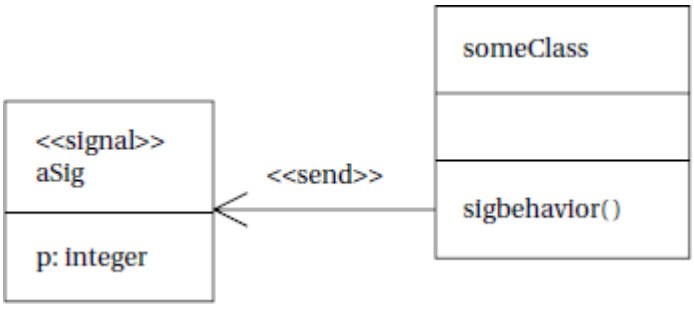


Fig Use of a UML signal.

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 3.11 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.

6. How do you evaluate operating system performance? (6) [CO3-H1]

- 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:
  - 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.
  - We have assumed that we know the execution time of the processes. In fact, we learned in Section 5.6 that program time is not a single number, but can be bounded by worst-case and best-case execution times.
  - 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 drastically degrade process execution time.
- The zero-time context switch assumption used in the analysis of RMS is not correct—we must execute instructions to save and restore context, and we must execute additional instructions to implement the scheduling policy. On the other hand, context switching can be implemented efficiently—context switching need not kill performance.
- The effects of nonzero context switching time must be carefully analyzed in the context of a particular implementation to be sure that the predictions of an ideal scheduling policy are sufficiently accurate.
- In most real-time operating systems, a context switch requires only a few hundred instructions, with only slightly more overhead for a simple real-time scheduler like RMS. When the overhead time is very small relative to the task periods, then the zero-time context switch assumption is often a reasonable approximation. Problems are most likely to manifest themselves in the highest-rate processes, which are often the most critical in any case.

- Completely checking that all deadlines will be met with nonzero context switching time requires checking all possible schedules for processes and including the context switch time at each preemption or process initiation. However, assuming an average number of context switches per process and computing CPU utilization can

provide at least an estimate of how close the system is to CPU capacity.

7. Explain the strategies of power optimization for processes. (16) [CO3-L1]

- The RTOS and system architecture can use static and dynamic power management mechanisms to help manage the system's power consumption. A **power management policy** [Ben00] is a strategy for determining 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.

  - Avoiding a power-down mode can cost unnecessary power.
  - Powering down too soon can cause severe performance penalties.

- 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, saving the requestor the start-up time. In general, predictive shutdown techniques are probabilistic they make guesses about activity patterns based on a probabilistic model of expected behaviour. Because they rely on statistics, they may not always correctly guess the time of the next activity.
- This can cause two types of problems:

- The requestor may have to wait for an activity period. In the worst case, the requestor may not make a deadline due to the delay incurred by system start-up.
- The system may restart itself when no activity is imminent. As a result, the system will waste power.

- Clearly, the choice of a good probabilistic model of service requests is important. The policy mechanism should also not be too complex, since the power it consumes to make decisions is part of the total system power budget.

- Several predictive techniques are possible. A very simple technique is to use fixed times. For instance, if the system does not receive inputs during an interval of length Ton, it shuts down; a powered-down system waits for a period T_{off} before returning to the power-on mode.

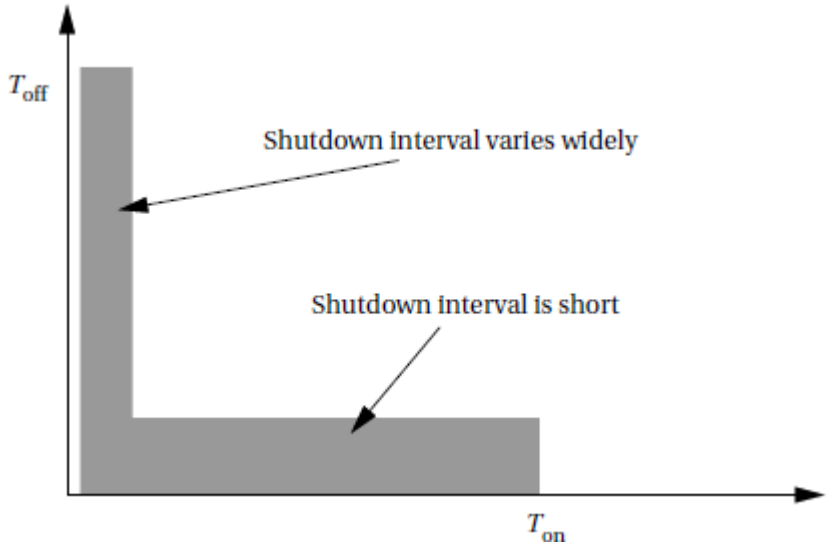


Fig An L-shaped usage distribution.

- The choice of T_{off} and Ton 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 . 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 industry 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 .

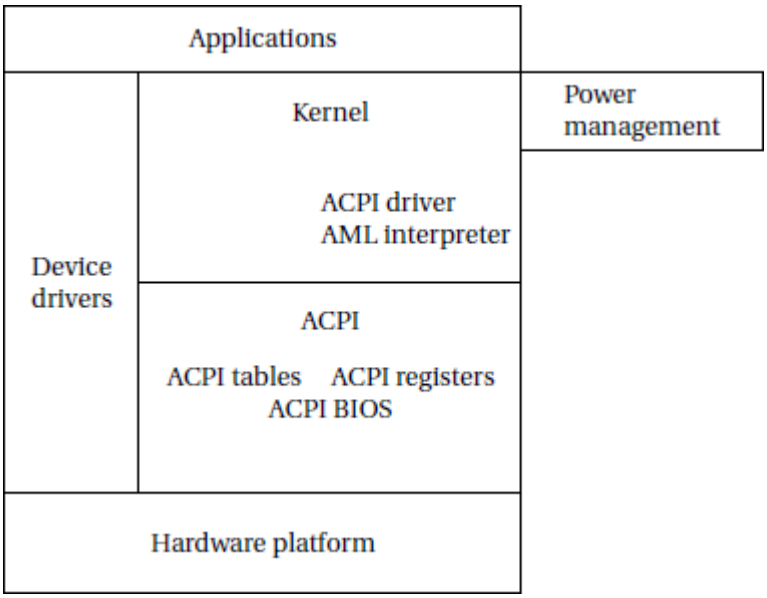


Fig The advanced configuration and power interface and its relationship to a complete system.

- ☐ 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:

- G3, the mechanical off state, in which the system consumes no power.
- G2, the soft off state, which requires a full OS reboot to restore the machine to working condition. This state has four substates:
- S1, a low wake-up latency state with no loss of system context;
- S2, a low wake-up latency state with a loss of CPU and system cache state;
- S3, a low wake-up latency state in which all system state except for main memory is lost; and S4, the lowest-power sleeping state, in which all devices are turned off.
- G1, the sleeping state, in which the system appears to be off and the time required to return to working condition is inversely proportional to power consumption.
- G0, the working state, in which the system is fully usable.
- The legacy state, in which the system does not comply with ACPI.

Unit – IV

System Design Techniques and Networks

Part – A

1. List the OSI layers from lowest to highest level of abstraction. [CO4-L1-April 2014]

• The OSI layers from lowest to highest level of abstraction are described below:

i. Physical layer
ii. Data link layer
iii. Network layer
iv. Transport layer
v. Session layer
vi. Presentation layer
vii. Application layer.

2. What is a distributed embedded architecture? [CO4-L1-April2014]

- ✓ In a distributed embedded system several process single elements are connected by a network that allows them to communicate.
- ✓ More than one computer or group of computer and share connected via network that forms distributed embedded systems.

3. What do you meant by accelerator/hardware accelerator? [CO4-L1-Nov/Dec 2013]

- ✓ An accelerator is one important category of processing element for embedded multiprocessor.
- ✓ It is attached to CPU buses to quickly execute certain key function.
- ✓ It provides large performance for many applications with computational kernels. It provides critical speedups for low-battery I/O functions.

5. What is the use of attached accelerator to CPU? [CO4-L1-May/June 2013]

- ✓ The CPU accelerator is attached to the CPU bus. TheCPU is also called the host. The CPU talks to the accelerator through the data and control registers in the accelerator.
- ✓ Control register allow the CPU to monitor the accelerator's operation and to give the accelerator commands.
- ✓ The CPU and accelerator will communicate via shared memory. The accelerator perform read and write operation directly.
- ✓ An accelerator interface is functionally equal to an I/O device but it does not perform input or output. CPUs and accelerators perform computations for specification.

6. What are the merits of embedded distributed architecture? [CO4-L1-Nov/Dec 2012]

- ✓ Error identification is more easy.
- ✓ It has more cost effective performance.
- ✓ Deadliness for processing the data are short.

7. What is the role played by the accelerator in the design of embedded system? [CO4-L1-Nov/Dec 2012]

- ✓ One important category of PE for embedded multiprocessor is the accelerator.
- ✓ An accelerator is attached to CPU buses to quickly execute certain key functions. It can provide large performance increase, for many applications with computational kernels.
- ✓ It can also provide critical speedups for low latency I/O functions.

8. Differentiate counter semaphore and binary semaphores. [CO4-L1-May/June 2012]

Counter semaphores	Binary semaphores
(i) Which allows an arbitrary resource count called counting.	(1) Which are restricted to values of 0 and 1 are called binary.
(ii) Synchronization of object that can have arbitrarily large number of states.	(2) Synchronization by two states (a) Not taken (b) Taken.

9. What is priority inheritance? [CO4-L1-May/June 2012]

Priority inheritance is a method of eliminating priority inversion, using this process scheduling algorithm will increase the priority of a process to the maximum priority of any process waiting for any resource on which the process has a resource lock.

10. Define Accelerators? [CO4-L1]

Hardware device or program designed to improve the overall performance of the computer. A good example is a 3D graphics accelerator, which has its own processor (GPU) and RAM, allowing it to perform tasks that would otherwise burden the other components of the computer. Another good example is a download accelerator, which can be installed to help improve download speeds.

11. What is meant by system? [CO4-L1]

A system is a collection of elements or components that are organized for a common purpose. A computer system consists of hardware components that have been carefully chosen so that they work well together and software components or programs that run in the computer.

12. What is network? [CO4-L1]

A network consists of two or more computers that are linked in order to share resources (such as printers and CDs), exchange files, or allow electronic communications. The computers on a network may be linked through cables, telephone lines, radio waves, satellites, or infrared light
beams.

13. What is a distributed embedded system? [CO4-L1]

Multiple computers interconnected by a network that share some common state and cooperate to achieve some common goal.

14. What is multistage network? [CO4-L1]

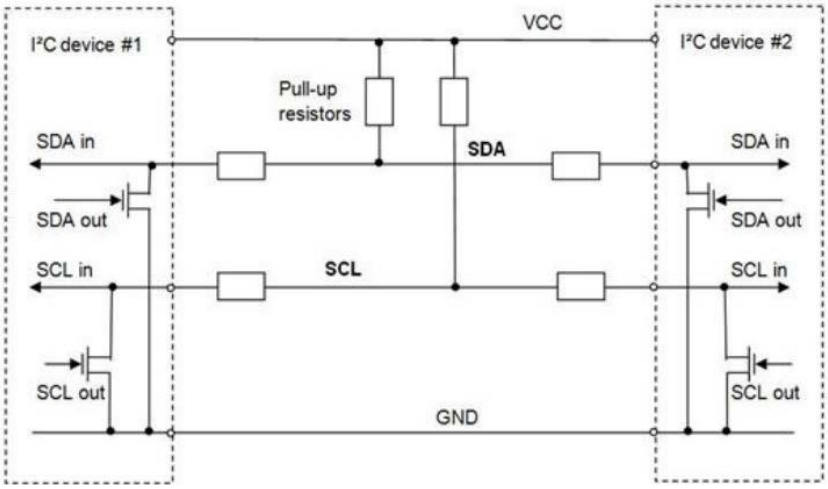
Multistage interconnection networks (MINs) are a class of highspeed computer networks usually composed of processing elements (PEs) on one end of the network andmemory elements (MEs) on the other end, connected by switching elements (SEs). The switching elements themselves are usually connected to each other in stages, hence the name.

15. Define CAN? [CO4-L1]

CAN bus (for controller area network) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer. CAN bus is a message-based protocol, designed specifically for automotive applications but now also used in other areas such as aerospace, maritime, industrial automation and medical equipment.

16. Draw the structure of an I²C bus? [CO4-L1]

Physically, the I²C bus consists of the 2 active wires SDA and SCL and a ground connection (refer to figure 4). The active wires are both bidirectional. The I2C protocol specification states that the IC that initiates a data transfer on the bus is considered the Bus Master. Consequently, at that time, all the other ICs are regarded to be Bus Slaves.



17. Define I²C bus? [CO4-L1]

I²C (Inter-Integrated Circuit), is a multi-master, multi-slave, single ended, serial computer bus invented by Philips Semiconductor, used for attaching low-speed peripherals to computer mother boards and embedded systems.

18. Define CSMA/CD? [CO4-L1]

Carrier Sense Multiple Access With Collision Detection (CSMA/CD) is a media access control method used most notably in local area networking using early Ethernet technology. It uses a carrier sensing scheme in which a transmitting data station detects other signals while transmitting a frame, and stops transmitting that frame, transmits a jam signal, and then waits for a random time interval before trying to resend the frame.

Part – B

1. Write short notes on Design methodologies. (6) [CO4-L1]

Process is important because without it, we can't reliably deliver the products we want to create. Thinking about the sequence of steps necessary to build some-thing may seem superfluous. But the fact is that everyone has their own design process, even if they don't articulate it. If you are designing embedded systems in your basement by yourself, having your own work habits is fine. But when several people work together on a project, they need to agree on who will do things and how they will get done. Being explicit about process is important when people work together. Therefore, since many embedded computing systems are too complex to be designed and built by one person, we have to think about design processes.

The obvious goal of a design process is to create a product that does some-thing useful. Typical specifications for a product will include functionality (e.g., cell phone), manufacturing cost (must have a retail price below $200), performance (must power up within 3 s), power consumption (must run for 12 h on two AA batteries), or other properties. Of course, a design process has several important goals beyond function,performance, and power.Three of these goals are summarized below.

Time-to-market*:* Customers always want new features. The product that comes out first can win the market, even setting customer preferences for future generations of the product. The profitable market life for some prod-ucts is 3–6 months—if you are 3 months late, you will never make money. In some categories,the competition is against the calendar,not just competitors. Calculators, for example, are disproportionately sold just before school starts in the fall. If you miss your market window,you have to wait a year for another sales season.

Design cost: Many consumer products are very cost sensitive. Industrial

buyers are also increasingly concerned about cost. The costs of designing the system are distinct from manufacturing cost—the cost of engineers' salaries, computers used in design, and so on must be spread across the units sold. In some cases, only one or a few copies of an embedded system may be built, so design costs can dominate manufacturing costs. Design costs can also be important for high-volume consumer devices when time-to-market pressures cause teams to swell in size.

Quality: Customers not only want their products fast and cheap, they also want them to be right. A design methodology that cranks out shoddy prod-ucts will soon be forced out of the marketplace. Correctness, reliability, and usability must be explicitly addressed from the beginning of the design job to obtain a high-quality product at the end.

Processes evolve over time. They change due to external and internal forces. Customers may change, requirements change, products change, and available com-ponents change. Internally, people learn how to do things better, people move on to other projects and others come in, and companies are bought and sold to merge and shape corporate cultures.

Software engineers have spent a great deal of time thinking about software design processes. Much of this thinking has been motivated by mainframe software such as databases. But embedded applications have also inspired some important thinking about software design processes.

A good methodology is critical to building systems that work properly. Delivering uggy systems to customers always causes dissatisfaction. But in some applications, such as medical and automotive systems, bugs create serious safety problems that can endanger the lives of users.

2. Explain the steps involved in designflow. (16) [CO4-L1]

A design flow is a sequence of steps to be followed during a design. Some of the steps can be performed by tools,such as compilers or CAD systems;other steps can be performed by hand. In this section we look at the basic characteristics of design flows.

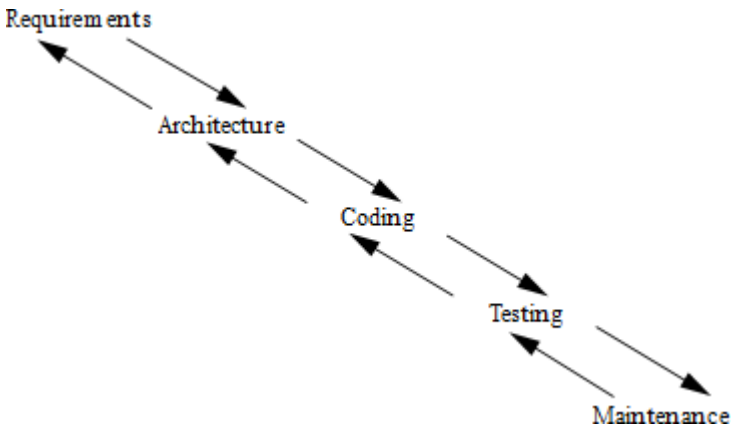
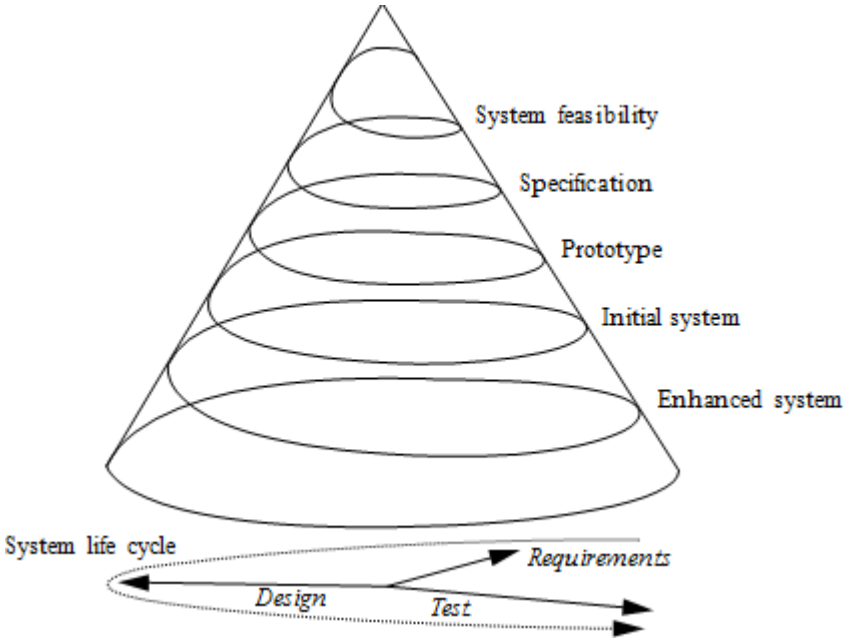


Figure **waterfall model**

Figure shows the **waterfall model** introduced by Royce [Dav90], the first model proposed for the software development process. The waterfall development model consists of five major phases: requirements analysis determines the basic characteristics of the system; architecture design decomposes the function-ality into major components; coding implements the pieces and integrates them; testing uncovers bugs; and maintenance entails deployment in the field, bug fixes, and upgrades. The waterfall model gets its name from the largely one-way flow of work and information from higher levels of abstraction to more detailed design steps (with a limited amount of feedback to the next-higher level of abstraction). Although top–down design is ideal since it implies good foreknowledge of the implementation during early design phases, most designs are clearly not quite so top–down. Most design projects entail experimentation and changes that require bottom–up feedback. As a result, the waterfall model is today cited as an unrealistic design process. However, it is important to know what the waterfall model is to be able to understand and how others are reacting against it.

Figure illustrates an alternative model of software development called the **spiral model** [Boe87]. While the waterfall model assumes that the system is built once in its entirety, the spiral model assumes that several versions of the system will be built. Early systems will be simple mock-ups constructed to aid designers' intuition and to build experience with the system. As design progresses,more com-plex systems will be constructed. At each level of design, the designers go through requirements,construction,and testing phases. At later stages when more complete versions of the system are constructed, each phase requires more work, widening the design spiral. This successive refinement approach helps the designers under-stand the system they are working on through a series of design cycles.



spiral model

The first cycles at the top of the spiral are very small and short, while the final cycles at the spiral's bottom add detail learned from the earlier cycles of the spiral. The spi-ral model is more realistic than the waterfall model because multiple iterations are often necessary to add enough detail to complete a design. However, a spiral methodology with too many spirals may take too long when design time is a major requirement.

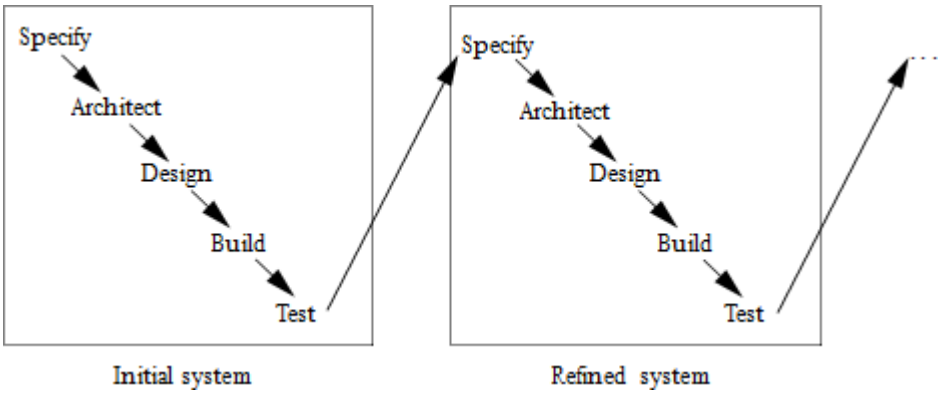


Figure **successive refinement**

Figure shows a **successive refinement** design methodology. In this approach, the system is built several times. A first system is used as a rough proto-type, and successive models of the system are further refined. This methodology makes sense when you are relatively unfamiliar with the application domain for which you are building the system. Refining the system

by building several increas-ingly complex systems allows you to test out architecture and design techniques. The various iterations may also be only partially completed;for example,continuing an initial system only through the detailed design phase may teach you enough to help you avoid many mistakes in a second design iteration that is carried through to completion.

Embedded computing systems often involve the design of hardware as well as software. Even if you aren't designing a board, you may be selecting boards and plugging together multiple hardware components as well as writing code.

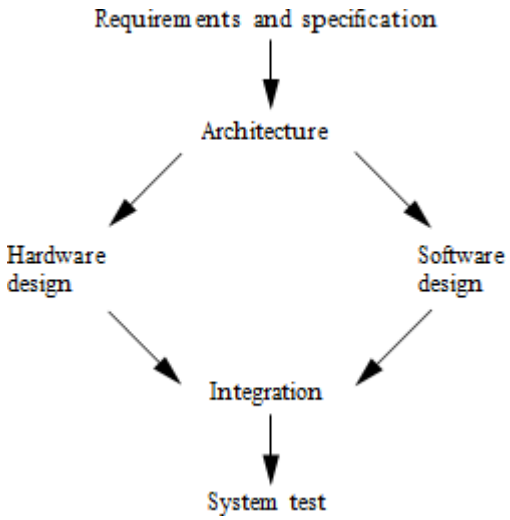
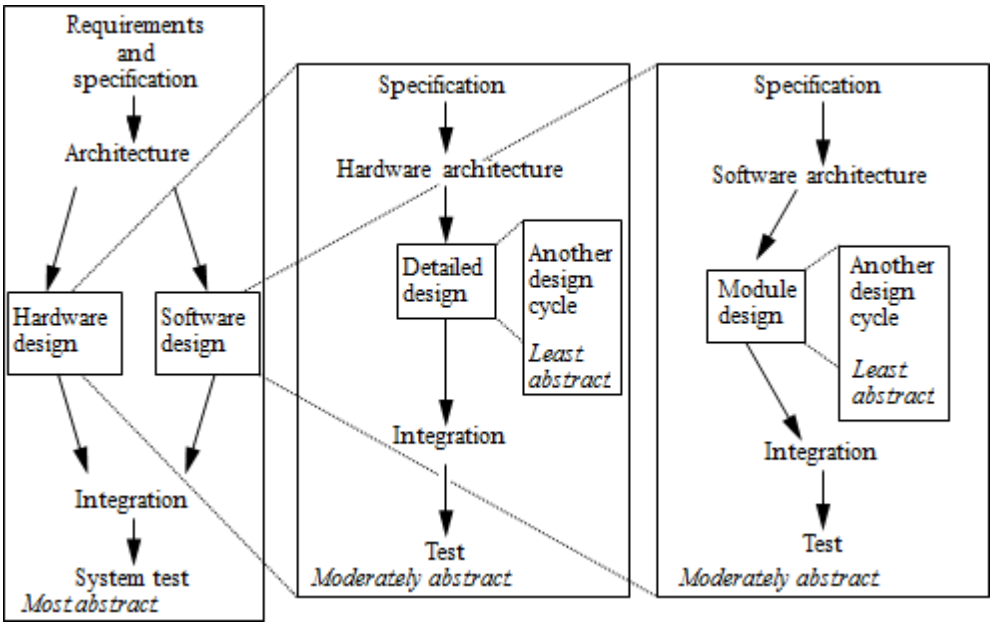


Figure shows a design methodology for a combined hardware/software project. Front-end activities such as specification and architecture simultaneously consider hardware and software aspects. Similarly, back-end integration and testing consider the entire system. In the middle, however, development of hardware and software components can go on relatively independently—while testing of one will require stubs of the other, most of the hardware and software work can proceed relatively independently.

In fact, many complex embedded systems are themselves built of smaller designs. The complete system may require the design of significant software com-ponents, custom logic, and so on, and these in turn may be built from smaller components that need to be designed. The design flow follows the levels of abstrac-tion in the system, from complete system design flows at the most abstract to design flows for individual components. The design flow for these complex sys-tems resembles the flow shown in Figure 9.5. The implementation phase of a flow is itself a complete flow from specification through testing. In such a large project, each flow will probably be handled by separate people or teams. The teams must rely on each other's results. The component teams take their requirements from the team handling the next higher level of abstraction, and the higher-level team relies on the quality of design and testing performed by the component team. Good communication is vital in such large

projects.

When designing a large system along with many people, it is easy to lose track of the complete design flow and have each designer take a narrow view of his or her role in the design flow. **Concurrent engineering** attempts to take a broader approach and optimize the total flow. Reduced design time is an important goal for concurrent engineering, but it can help with any aspect of the design that cuts across the design flow, such as reliability, performance, power consumption, and so on. It tries to eliminate "over-the-wall" design steps, in which one designer performs an isolated task and then throws the result over the wall to the next designer, with little interaction between the two. In particular, reaping the most benefits from concurrent engineering usually requires eliminating the wall betwee n design and manufacturing. Concurrent engineering efforts are comprised of several elements.



Cross-functional teams include members from various disciplines involved in the process, including manufacturing, hardware and software design, marketing, and so forth.

Concurrent product realization process activities are at the heart of concurrent engineering. Doing several things at once, such as designing various subsystems simultaneously, is critical to reducing design time.

Incremental information sharing and use helps minimize the chance that concurrent product realization will lead to surprises. As soon as new information becomes available, it is shared and integrated into the design. Cross-functional teams are important to the effective sharing of information in a timely fashion.

Integrated project management ensures that someone is responsible for the entire project, and that responsibility is not abdicated once one aspect of the work is done.

Early and continual supplier involvement helps make the best use of suppliers' capabilities

Early and continual customer focus helps ensure that the product best meets customers' needs

3. Explain the requirement analysis. (6) [CO4-L1]

Requirements are informal descrip-tions of what the customer wants,while **specifications** are more detailed,precise, and consistent descriptions of the system that can be used to create the architec-ture. Both requirements and specifications are, however, directed to the outward behavior of the system, not its internal structure.

The overall goal of creating a requirements document is effective communication between the customers and the designers. The designers should know what they are expected to design for the customers; the customers, whether they are known in advance or represented by marketing, should understand what they will get.

We have two types of requirements: **functional** and **nonfunctional**. A func-tional requirement states what the system must do, such as compute an FFT. A nonfunctional requirement can be any number of other attributes, including physical size, cost, power consumption, design time, reliability, and so on.

A good set of requirements should meet several tests [Dav90]:

Correctness: The requirements should not mistakenly describe what the customer wants. Part of correctness is avoiding over-requiring—the require-ments should not add conditions that are not really necessary.

Unambiguousness: The requirements document should be clear and have only one plain language interpretation

Completeness: All requirements should be included

Verifiability: There should be a cost-effective way to ensure that each require-ment is satisfied in the final product. For example, a requirement that the system package be "attractive" would be hard to verify without some agreed upon definition of attractiveness

Consistency: One requirement should not contradict another requirement

Modifiability: The requirements document should be structured so that it can be modified to meet changing requirements without losing consistency, verifiability, and so forth.

Traceability: Each requirement should be traceable in the following ways

We should be able to trace backward from the requirements to know why each requirement exists.

We should also be able to trace forward from documents created before the requirements (e.g., marketing memos) to understand how they relate to the final requirements.

We should be able to trace forward to understand how each requirement is satisfied in the implementation.

We should also be able to trace backward from the implementation to know which requirements they were intended to satisfy.

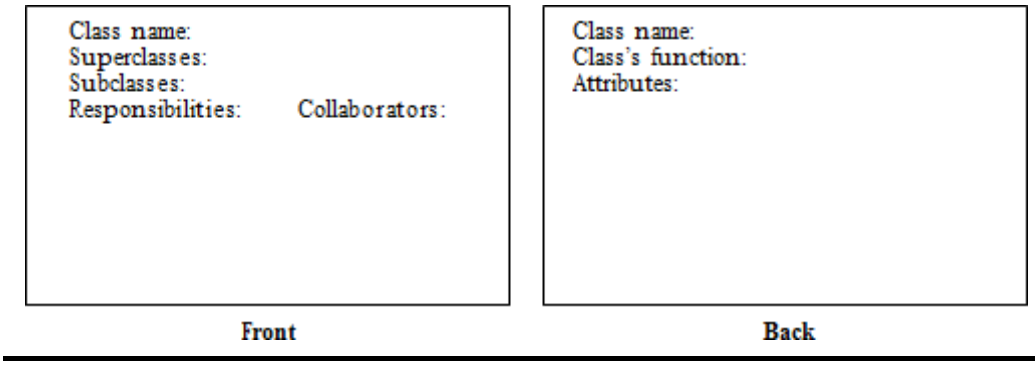
How do you determine requirements? If the product is a continuation of a series, then many of the requirements are well understood. But even in the most modest upgrade, talking to the customer is valuable. In a large company, marketing or sales departments may do most of the work of asking customers what they want,but a surprising number of companies have designers talk directly with customers.Direct customer contact gives the designer an unfiltered sample of what the customer says. It also helps build empathy with the customer,which often pays off in cleaner, easier-to-use customer interfaces.Talking to the customer may also include conduct-ing surveys,organizing focus groups,or asking selected customers to test a mock-up or prototype.

4. Describe system analysis and architecture design. (8) [CO4-H1]

we look at how to get a handle on the overall system architecture. The **CRC card** methodology is a well-known and useful way to help analyze a system's structure. It is particularly well suited to object-oriented design since it encourages the encapsulation of data and functions. The acronym CRC stands for the following three major items that the methodology tries to identify:
Classes define the logical groupings of data and functionality.
Responsibilities describe what the classes do.
Collaborators are the other classes with which a given class works.

The name CRC card comes from the fact that the methodology is practiced by
having people write on index cards. (In the United States,the standard size for index cards is 3 _ 5__, so these cards are often called 3 _ 5 cards.) An example card is shown in Figure 9.10; it has space to write down the class name, its responsibilities and collaborators, and other information. The essence of the CRC card methodology is to have people write on these cards, talk about them, and update the cards until they are satisfied with the results.

This technique may seem like a primitive way to design computer systems. However, it has several important advantages. First, it is easy to get non computer people to create CRC cards. Getting the advice of domain experts (automobile designers for automotive electronics or human factors experts for PDA design, for example) is very important in system design. The CRC card methodology is informal enough that it will not intimidate non-computer specialists and will allow you to capture their input. Second, it aids even computer specialists by encouraging



them to work in a group and analyze scenarios. The walkthrough process used with CRC cards is very useful in scoping out a design and determining what parts of a system are poorly understood. This informal technique is valuable to tool-based design and coding. If you still feel a need to use tools to help you practice the CRC methodology, software engineering tools are available that automate the creation of CRC cards.

Before going through the methodology, let's review the CRC concepts in a little more detail. We are familiar with classes—they encapsulate functionality. A class may represent a real-world object or it may describe an object that has been created solely to help architect the system. A class has both an internal state and a functional interface the functional interface describes the class's capabilities. The responsibility set is an informal way of describing that functional interface. The responsibilities provide the class's interface, not its internal implementation. Unlike describing a class in a programming language, however, the responsibilities may be described informally in English (or your favourite language). The collaborators of a class are simply the classes that it talks to, that is, classes that use its capabilities or that it calls upon to help it do its work.

The class terminology is a little misleading when an object-oriented programmer looks at CRC cards. In the methodology, a class is actually used more like an object in an OO programming language—the CRC card class is used to represent a real actor in the system. However, the CRC card class is easily transformable into a class dentition in an object-oriented design.

CRC card analysis is performed by a team of people. It is possible to use it by yourself, but a lot of the benefit of the method comes from talking about the devel-oping classes with others. Before becoming the process, you should create a large number of CRC cards using the basic format shown in Figure 9.10. As you are work-ing in your group, you will be writing on these cards; you will probably discard many of them and rewrite them as the system evolves. The CRC card method-ology is informal, but you should go through the following steps when using it to analyze a system:

5. Discus about Quality assurance technique. (6) [CO4-L3]

The International Standards Organization (ISO) has created a set of quality stan-dards known as **ISO 9000**. ISO 9000 was created to apply to a

broad range of industries, including but not limited to embedded hardware and software. A stan-dard developed for a particular product, such as wooden construction beams,could specify criteria particular to that product, such as the load that a beam must be able to carry. However, a wide-ranging standard such as ISO 9000 cannot specify the detailed standards for every industry. Consequently, ISO 9000 concentrates on pro-cesses used to create the product or service. The processes used to satisfy ISO 9000 affect the entire organization as well as the individual steps taken during design and manufacturing.

A detailed description of ISO 9000 is beyond the scope of this book; several books [Sch94, Jen95] describe ISO 9000's applicability to software development. We can, however, make the following observations about quality management based on ISO 9000:

Process is crucial: Haphazard development leads to haphazard products and low quality. Knowing what steps are to be followed to create a high-quality product is essential to ensuring that all the necessary steps are in fact followed.

Documentation is important: Documentation has several roles:

The creation of the documents describing processes helps those involved understand the processes documentation helps internal quality monitoring groups to ensure that the required processes are actually being followed; and documentation also helps outside groups (customers,auditors,etc.) understand the processes and how they are being implemented.

Communication is important: Quality ultimately relies on people. Good documentation is an aid for helping people understand the total quality process. The people in the organization should understand not only their specific tasks but also how their jobs can affect overall system quality.

Many types of techniques can be used to verify system designs and ensure quality. Techniques can be either manual or tool based. Manual techniques are surprisingly effective in practice. In Section 9.5.3 we discuss design reviews, which are simply meetings at which the design is discussed and which are very successful in identifying bugs. Many of the software testing techniques described in Section 5.10 can be applied manually by tracing through the program to determine the required tests. Tool-based verification helps considerably in managing large quantities of information that may be generated in a complex design. Test generation programs can automate much of the drudgery of creating test sets for programs. Tracking tools can help ensure that various steps have been performed. Design flow tools automate the process of running design data through other tools.

Metrics are important to the quality control process. To know whether we have achieved high levels of quality, we must be able to measure aspects of the system and our design process. We can measure certain aspects of the system itself, such as the execution speed of programs or the coverage of test

patterns. We can also measure aspects of the design process, such as the rate at which bugs are found. Section describes ways in which measurements can be used in the QA process.

Tool and manual techniques must fit into an overall process. The details of that process will be determined by several factors, including the type of product being designed (e.g., video game, laser printer, air traffic control system), the number of units to be manufactured and the time allowed for design, the existing practices in the company into which any new processes must be integrated, and many other factors. An important role of ISO 9000 is to help organizations study their total process, not just particular segments that may appear to be important at a particular time.

One well-known way of measuring the quality of an organization's software development process is the Capability Maturity Model (CMM) developed by Carnegie Mellon University's Software Engineering Institute [SEI99]. The CMM provides a model for judging an organization. It defines the following five levels of maturity:

1. Initial: A poorly organized process, with very few well-defined processes. Success of a project depends on the efforts of individuals,not the organization itself.
2. Repeatable: This level provides basic tracking mechanisms that allow man-agement to understand cost, scheduling, and how well the systems under development meet their goals.
3. Defined: The management and engineering processes are documented and standardized. All projects make use of documented and approved standard methods.
4. Managed: This phase makes detailed measurements of the development process and product quality.
5. Optimizing: At the highest level, feedback from detailed measurements is used to continually improve the organization's processes.

The Software Engineering Institute has found very few organizations anywhere in the world that meet the highest level of continuous improvement and quite a few organizations that operate under the chaotic processes of the initial level. However, the CMM provides a benchmark by which organizations can judge themselves and use that information for improvement.

6. Explain about distributed embedded architecture. (6) [CO4-L1]

- A distributed embedded system can be organized in many different ways, but its basic units are the PE and the network as illustrated in Figure 4.4. A PE may be an instruction set processor such as a DSP, CPU, or microcontroller, as well as a nonprogrammable unit such as the ASICs used to implement PE 4. An I/O device such as PE 1 (which we call here a **sensor** or **actuator**, depending on whether it provides input or output) may also be a PE, so long as it can speak the network protocol to communicate with other PEs.

- The network in this case is a bus, but other network topologies are also

possible. It is also possible that the system can use more than one network, such as when relatively independent functions require relatively little communication among them. We often refer to the connection between PEs provided by the network as a **communication link**.

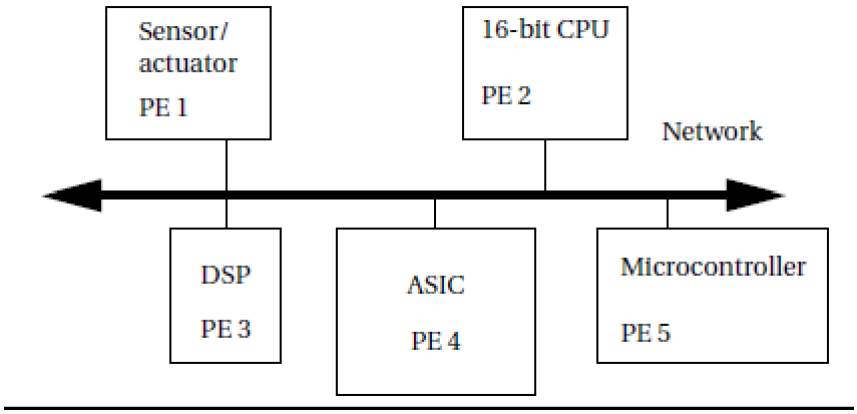


Fig An example of a distributed embedded system.

☐ The system of PEs and networks forms the **hardware platform** on which the application runs.

☐ In particular, PEs do not fetch instructions over the network as they do on the microprocessor bus. We take advantage of this fact when analyzing network performance the speed at which PEs can communicate over the bus would be difficult if not impossible to predict if we allowed arbitrary instruction and data fetches as we do on microprocessor buses.

Distributed

☐ Building an embedded system with several PEs talking over a network is definitely more complicated than using a single large microprocessor to perform the same tasks. So why would anyone build a distributed embedded system? All the reasons for designing accelerator systems also apply to distributed embedded systems, and several more reasons are unique to distributed systems.

☐ In some cases, distributed systems are necessary because the devices that the PEs communicate with are physically separated. If the deadlines for processing the data are short, it may be more cost-effective to put the PEs where the data are located rather than build a higher-speed network to carry the data to a distant, fast PE.

☐ An important advantage of a distributed system with several CPUs is that one part of the system can be used to help diagnose problems in another part. Whether you are debugging a prototype or diagnosing a problem in the field, isolating the error to one part of the system can be difficult when everything is done on a single CPU. If you have several CPUs in the system, you can use

one to generate inputs for another and to watch its output.

Network Abstractions

- Networks are complex systems. Ideally, they provide high-level services while hiding many of the details of data transmission from the other components in the system. In order to help understand (and design) networks, the International Standards Organization has developed a seven-layer model for networks known as Open Systems Interconnection (OSI) models [Sta97A]. Understanding the OSI layers will help us to understand the details of real networks.

- The seven layers of the **OSI model**, shown in Figure 4.5, are intended to cover a broad spectrum of networks and their uses. Some networks may not need the services of one or more layers because the higher layers may be totally missing or an intermediate layer may not be necessary.

- However, any data network should fit into the OSI model. The OSI layers from lowest to highest level of abstraction are described below.

  - Physical: The physical layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires), electrical properties, basic functions of the electrical and physical components, and the basic procedures for exchanging bits.

  - Data link: The primary purpose of this layer is error detection and control across a single link. However, if the network requires multiple hops over several data links, the data link layer does not define the mechanism for data integrity between hops, but only within a single hop.

  - Network: This layer defines the basic end-to-end data transmission service. The network layer is particularly important in multi hop networks.

  - Transport: The transport layer defines connection-oriented services that ensure that data are delivered in the proper order and without errors across multiple links. This layer may also try to optimize network resource utilization.

Application	End-use interface
Presentation	Data format
Session	Application dialog control
Transport	Connections
Network	End-to-end service
Data link	Reliable data transport
Physical	Mechanical, electrical

Fig The OSI model layers.

- Session: A session provides mechanisms for controlling the interaction of end user services across a network, such as data grouping and check pointing.

- Presentation: This layer defines data exchange formats and provides transformation utilities to application programs.

- Application: The application layer provides the application interface between the network and end-user programs.

☐ Although it may seem that embedded systems would be too simple to require use of the OSI model, the model is in fact quite useful.

☐ Even relatively simple embedded networks provide physical, data link, and network services.

An increasing number of embedded systems provide Internet service that requires implementing the full range of functions in the OSI model.

7. Write short notes on multiprocessor environment. (6) [CO4-L1]

Programming a single CPU is hard enough. Why make life more difficult by adding more processors? A multiprocessor is, in general, any computer system with two or more processors coupled together. Multiprocessors used for scientific or business applications tend to have regular architectures: several identical proces-sors that can access a uniform memory space. We use the term processing element (PE) to mean any unit responsible for computation, whether it is programmable or not.

Embedded system designers must take a more general view of the nature of multiprocessors. As we will see, embedded computing systems are built on top of an astonishing array of different multiprocessor architectures.

Why is there no single multiprocessor architecture for all types of embedded

computing applications? And why do we need embedded multiprocessors at all? The reasons for multiprocessors are the same reasons that drive all of embedded system design: real-time performance, power consumption, and cost.

The first reason for using an embedded multiprocessor is that they offer significantly better cost/performance—that is, performance and functionality per dollar spent on the system—
than would be had by spending the same amount of money on a uniprocessor system.The basic reason for this is that processing element purchase price is a nonlinear function of performance [Wol08]. The cost of a microproces-sor increases greatly as the clock speed increases. We would expect this trend as a normal consequence of VLSI fabrication and market economics. Clock speeds are normally distributed by normal variations in VLSI processes; because the fastest chips are rare, they naturally command a high price in the marketplace.

Because the fastest processors are very costly, splitting the application so that it can be performed on several smaller processors is usually much cheaper. Even with the added costs of assembling those components, the total system comes out to be less expensive. Of course, splitting the application across multiple processors does entail higher engineering costs and lead times, which must be factored into the project.

In addition to reducing costs, using multiple processors can also help with real-time performance. We can often meet deadlines and be responsive to interaction much more easily when we put those time-critical processes on separate processors. Given that scheduling multiple processes on a single CPU incurs overhead in most realistic scheduling models,as discussed in Chapter 6,putting the time-critical processes on PEs that have little or no time-sharing reduces scheduling overhead. Because we pay for that overhead at the nonlinear rate for the processor, as illus-tratedin Figurethesavingsbysegregatingtime-critical processescan belarge—it may take an extremely large and powerful CPU to provide the same responsiveness that can be had from a distributed system.

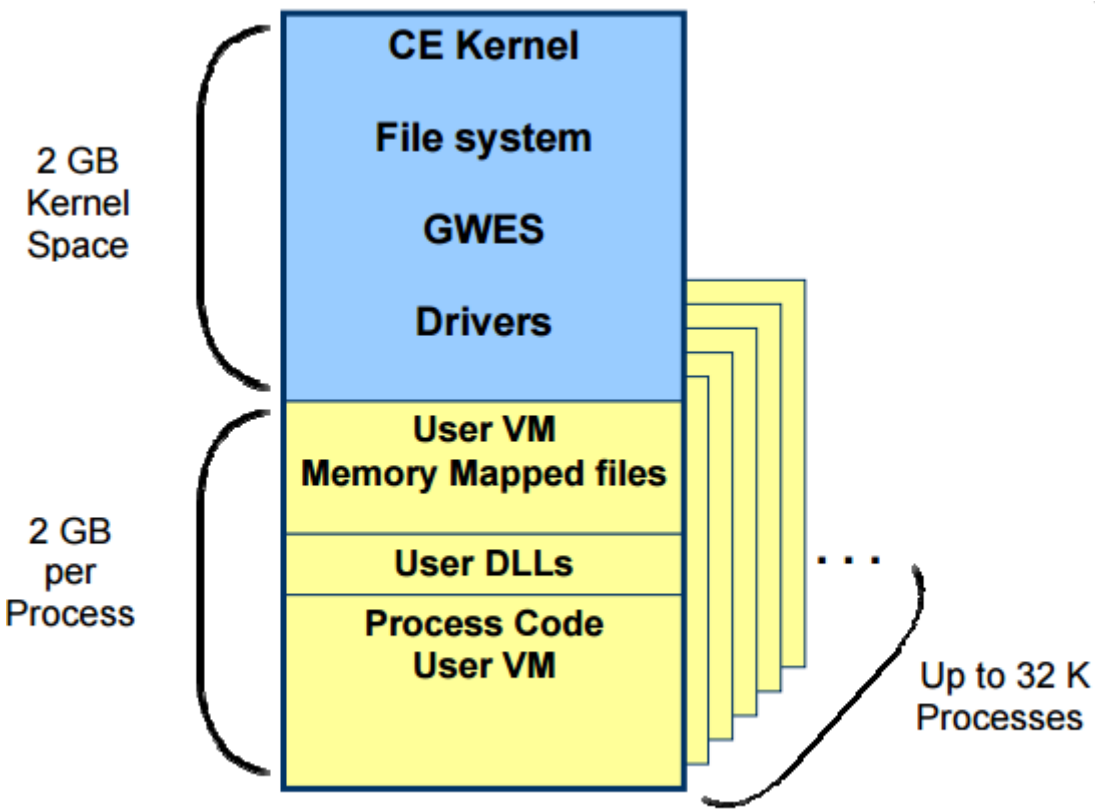
Many of the technology trends that encourage us to use multiprocessors for performance also lead us to multiprocessing for low power embedded computing. Several processors running at slower clock rates consume less power than a single large processor: performance scales linearly with power supply voltage but power scales with V^2.

8. Explain in detail about Windows CE real time operating system and its architecture. (16) [CO4-H1]

To better understand the features and capabilities of a Real-Time operating Fig system (RTOS), we will examine the architecture and features of a commercial RTOS designed for use in embedded systems. It is assumed that the reader already has some familiarity with general purpose operating systems. General purpose operating systems such as the ones typically used in desktop PCs do not provide the real-time response rates needed, require more hardware support, and higher power levels than are typically found in the majority of current embedded devices. Windows Embedded CE is a

popular commercial real-time operating system used in many embedded devices. CE is not just a modified version of the Windows desktop operating systems; it is a totally different OS. It was originally developed from scratch starting in the mid 1990s to provide a real time OS for embedded devices with less memory and processor power than a desktop PC. CE is also the core technology behind Windows Mobile devices including the Smart Phone and Pocket PC.

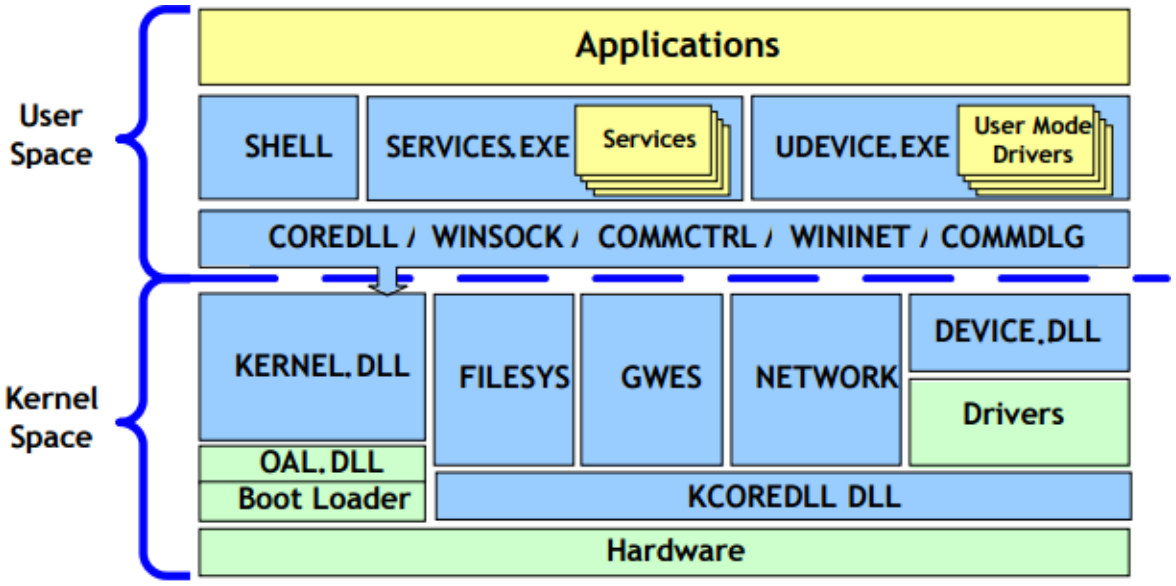
We will use CE as the OS for our eBox 2300 embedded computer system in laboratory projects. In this chapter, we will limit our discussions to CE 6.0, the newest version. Real-time response times are faster and kernel size is typically smaller than a general purpose OS. CE 6.0 can run a maximum of 32,000 simultaneous processes, each with 2 GB of virtual memory 6.1. Recall that the amount of physical memory present in a computer ependent of the virtual memory address space (a device runn d 4G of Physical RAM). ludes C/C++ and C# cross compilers, a build system, and extensive tools that run on a desktop PC. Th several games are also included. The desktop PC is typically connected to the target system for development work. A special GUI-based tool called Platform Builder is used to generate new OS kernels. Several different families of popular embedded processors are supported including X86, ARM, SHx, MIPS. The CE system architecture is shown in Figure 6.2. Items in blue are provided with the OS, the OEM typically provides the items shown in green, and applications programs provide the items shown in yellow.



The Windows Embedded CE 6.0 Virtual Memory Space Model.

The CE architecture is designed around two modes of privilege, Kernel and User. The fundamental system components providing basic services to the operating system

run in the privileged Kernel mode, while user processes (applications) and dlls run in the unprivileged User mode. Privileged components include the kernel, device manager, file system manager, GWES etc. The privileged kernel components are always present no matter what user ponent often residing in the kernel. Threads must to process t buffers pas ust be validated and ccess which also requires kernel overhead. Component ot incur the overhead of a kernel ments because they are always present and already privileged. Calls between kernel mode components el intervention. Many API calls require



Windows Embedded CE 6.0 Architecture.

support from a number of different system servers (filesys, device, kernel, etc) to complet rs have been moved from separate ser mode processes into the kernel in CE6, eliminating much of the overhead. his change is responsible for significant performance increases in many d in C/C++ or C#. Windows CE I and Component) offers a standard for creating components that can be into larger systems. COM defines binary objects that can cation framework. The Active Template process happens to be running. A single user mode process is resident at one time, isolated from other user mode processes and from the kernel. Threads running in user mode processes make system API calls that are handled by another com migrate to the server process residing in the kernel or another user process complete the API call. This transition out of the unprivileged user mode in o another process requires expensive kernel support. Memory sed between processes or into the kernel m marshaled for proper a s running in the privileged kernel mode do n trap and some of the data marshalling require are done directly without kern e a single call. These system serve u T system API calls compared to previous versions of the operating system.

9. Explain the POSIX real time operating system. (10) [CO4-H1]

Real-time Systems A real-time system is one where the timeliness of the result of a calculation is important]. Examples include military weapons systems, factory control systems, and Internet video and audio streaming. Real time systems are typically categorized into two classes: hard and soft. In a hard real-time system the time deadlines must be met or the result of a calculation is invalid. For example in a missile tracking system, if the missile is delayed it may miss its intended target. The timing constraints in a soft real-time system are not as stringent. There is still some utility to the result of a calculation if it does not meet its timing deadline. Internet audio/video streaming is an example of a soft real-time system. If a packet of data is late or lost the quality of the audio/video is degraded, but the stream may still be audible.

POSIX profiles Embedded systems typically have space and resource limitations, and an operating system that includes all the features of POSIX may not be appropriate. The POSIX 1003.13 profile standard was defined to address these types of systems. POSIX 1003.13 does not contain any additional features; instead it groups the functions from existing POSIX standards into units of functionality. The profiles are based on whether or not an operating system supports more than one process and a file system. The four current profiles are summarized in Table.

Table POSIX 1003.13 Profiles

Profile	Number of Processes	Threads	File System
54	Multiple	Yes	Yes
53	Multiple	Yes	No
52	Single	Yes	Yes
51	Single	Yes	No

POSIX real-time extensions

POSIX 1003.1b, as well as 1003.1d and 1003.1j define extensions useful for development of real-time systems. Functions defined in the original real-time extension standard 1003.1b are supported across a wider number of operating systems than the other two specifications. For this reason this paper focuses on POSIX 1003.1b.

The following features constitute the bulk of the features defined in POSIX 1003.1b:

• Timers: Periodic timers, delivery is accomplished using POSIX signals

• Priority scheduling: Fixed priority preemptive scheduling with a minimum of 32 priority levels

• Real-time signals: Additional signals with multiple levels of priority • Semaphores: Named and memory counting semaphores

• Memory queues: Message passing using named queues

• Shared memory: Named memory regions shared between multiple processes

• Memory locking: Functions to prevent virtual memory swapping of physical memory pages.

POSIX threads

In POSIX, threads are implemented in an independent specification, which means that their specification is independent of the other real-time features . Because of this there are a number of features from the real-time specification that are carried over to the thread specification. For example priority scheduling is done on a per-thread basis, but is handled in a similar manner as scheduling in POSIX 1003.1b. A thread's priority and scheduling policy is typically specified when it is created. The POSIX thread specification defines functionality and/or makes modifications to POSIX in the following areas:

• Thread control: Creation, deletion and management of individual threads

• Priority scheduling: POSIX real-time scheduling extended to include scheduling on a per thread basis; the scheduling scope is either done globally across all threads in all processes, or performed locally within each process

• Mutexes: Used to guard critical sections of code; mutexes also include support for priority inheritance and priority ceiling protocols to help prevent priority inversions

• Condition variables: Used in conjunction with mutexes, condition variables can be used to create a monitor synchronization structure

• Signals: Ability to deliver signals to individual threads

Unit – V

Case Study

Part - A

1. What is a PDA? [CO5-L1-April 2014]

PDA (Personal Digital Assistant) is a device that can be used to receive, display and transcribe information. PDA can run a wide variety of applications.

2. What is a set-top box or STB or STU? [CO5-L1-April 2014]

A set top box (STB) or set top unit (STU) is an information appliance device that generally contains a tuner and connects to a television set and an external source of signal, turning the source signal into content in a form that can then be displayed on the television screen or other display device.

USES :

a) Cable television and satellite television system.

3. Write short notes on H/W and S/W co-design. [CO5-L1-Nov/Dec 2013]

Embedded systems architecture design is the task of selecting and programming a suitable configuration of components for a required system application. Building an embedded system is not an easy task. Every embedded system consist of an embedded hardware and embedded software.

So software and hardware plays a main role in design of embedded system architecture.

Need For Co-Design :

- ✓ Co-design refers to parallel or concurrent development of hardware and software for an embedded system.
- ✓ Co-design reduces the overall design and development cycle of the embedded system.
- ✓ It helps the designer to find the bugs at early stage.
- ✓ It also reduces the number of errors, particularly at the hardware-software interface level.

4. What are FOSS tools for embedded systems? [CO5-L1-Nov/Dec 2013 & May/June 2013]

GNU Compiler Collection (GCC) and GNU debugger (GDB) are the most popular FOSS (Free and open source) tools used in embedded systems.

5. List the major components in the Personal Digital Assistant System? [CO5-L1-May/June 2013]

- ✓ Process or memory
- ✓ Connectivity
- ✓ Power management unit
- ✓ User interface.

6. Define Data Compression. [CO5-L1]

Reducing the 'electronic space' (data bits) used in representing a piece of information, by eliminating the repetition of identical sets of data bits (redundancy) in an audio/video, graphic, or text data file.

7. Why most designers use FOSS tools in embedded system development? [CO5-H1-Nov/Dec 2012]

Because,

- ✓ It makes software portable.
- ✓ It speeds up the development process
- ✓ It provides good foundation for system development activities.

8. What is signal servicing function? [CO5-L1-May/June 2012]

The signal service is a bureau of the government organized to collect from the whole country simultaneously report to local metrologicalcondition upon comparison of which at certain office, predictions concerning the weather are telegraphed to various sections also known as signal publicity display.

9. Give the steps to destroy a message queue. [CO5-L1-May/June 2012]

- ✓ First delete all the element in a message queue.
- ✓ Check if Front = rear = -1, then queue is empty.
- ✓ Otherwise, now call a delete routine to destroy a message queue.

10. Define SOC? [CO5-L1]

Embedded systems are being designed on a single silicon chip called system on chip. SOC is a new design innovation for embedded system Ex. Mobile phone.

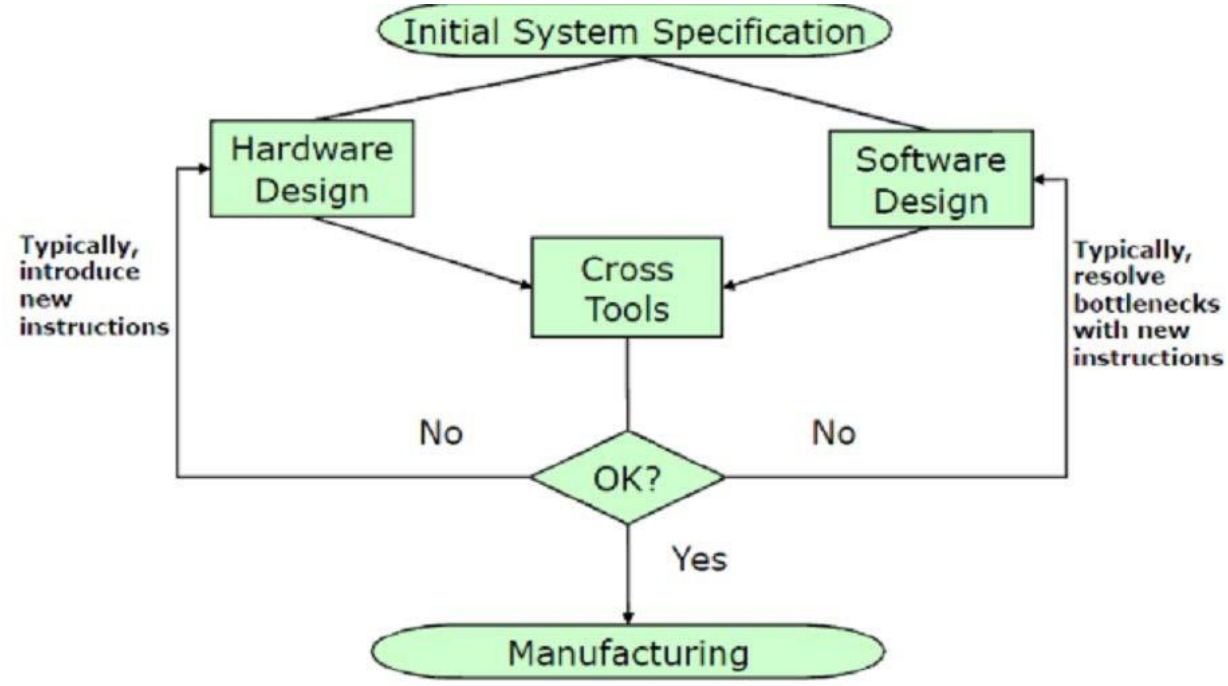
11. Define PDA? [CO5-L1]

A personal digital assistant (PDA), also known as a palmtop computer, or personal data assistant, is a mobile device that functions as a personal information manager. The term evolved from Personal Desktop Assistant, a software term for an application that prompts or prods the user of a computer with suggestions or provides quick reference to contacts and other lists. PDAs were discontinued in early 2010s after the widespread adoption
of smart phones.

12. Define software MODEM. [CO5-L1]

A software modem is a low-cost alternative to a standard hardware-based modem. While hardware modems contain all the parts necessary to connect to the internet, the software version transfers some of that work to the computer's processor.

13. Draw the hardware and Software Co-design? [CO5-L1]



Part – B

1. Design a data compressor. (16) [CO5-H3]

Our design example for this chapter is a data compressor that takes in data with a constant number of bits per data element and puts out a compressed data stream in which the data is encoded in variable-length symbols. Because this chapter concentrates on CPUs,we focus on the data compression routine itself.

Requirements and Algorithm

We require some understanding of how our compression code fits into a larger system. Figure shows a collaboration diagram for the data compression process.The data compressor takes in a sequence of **input symbols** and then produces a stream of **output symbols**. Assume for simplicity that the input symbols are one byte in length.The output symbols are variable length,sowe have to choose a format in which to deliver the output data. Delivering each coded symbol separately is tedious, since we would have to supply the length of each symbol and use external code to pack them into words. On the other hand, bit-by-bit delivery is almost certainly too slow. Therefore,we will rely on the data compressor to pack the coded symbols into an array.

There is not a one-to-one relationship between the input and output symbols,and we may have to wait for several input symbols before a packed
output word comes out.

Huffman coding for text compression

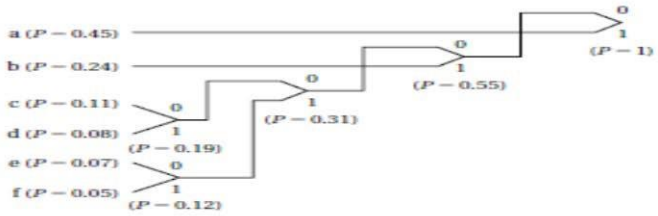
Text compression algorithms aim at statistical reductions in the volume of data. One commonly used compression algorithm is Huffman coding [Huf52], which makes use of information

on the frequency of characters to assign variable-length codes to characters. If shorter bit sequences are used to identify more frequent characters, then the length of the total sequence will be reduced.

In order to be able to decode the incoming bit string, the code characters must have unique prefixes: No code may be a prefix of a longer code for another character. As a simple example of Huffman coding, assume that these characters have the following probabilities P of appearance in a message:

Character	P	Character	P
A	0.45	D	0.08
B	0.24	E	0.07
C	0.11	F	0.05

We build the code from the bottom up. After sorting the characters by probability, we create a new symbol by adding a bit. We then compute the joint probability of finding either one of those characters and re-sort the table. The result is a tree that we can read top down to find the character codes. The coding tree for our example appears



Reading the codes off the tree from the root to the leaves, we obtain the following coding of the characters:

Character	Code	Character	Code
A	1	D	0001
B	01	E	0010
C	0000	F	0011

below.

Once the code has been constructed, which in many applications is done off-line,the codes can be stored in a table for encoding. This makes encoding simple, but clearly the encoded bit rate can vary significantly depending on the input character sequence. On the decoding side, since we do not know a priori the length of a character's bit sequence, the computation time required to decode a character can vary significantly.The data compressor as discussed above is not a complete system, but we can create at least a partial requirements list for the module as seen below.We used the

abbreviation N/A for not applicable to describe some items that do not make sense for a code module.

Name	Data compression module
Purpose	Code module for Huffman data compression
Inputs	Encoding table, uncoded byte-size input symbols
Outputs	Packed compressed output symbols
Functions	Huffman coding
Performance	Requires fast performance
Manufacturing cost	N/A
Power	N/A
Physical size and weight	N/A

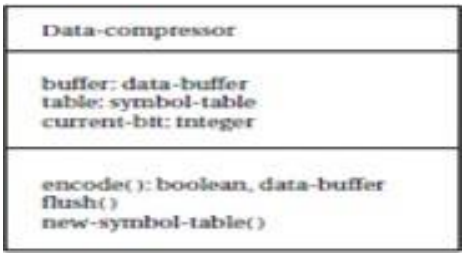
Specification

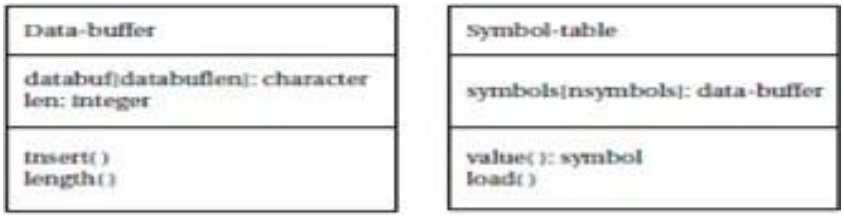
Let's refine the description of Figure 3.20 to come up with a more complete specification for our data compression module.That collaboration diagram concentrates on the steady-state behavior of the system. For a fully functional system,we have to provide the following additional behavior.

- We have to be able to provide the compressor with a new symbol table.
- We should be able to flush the symbol buffer to cause the system to release all pending symbols that have been partially packed.We may want to do this when we change the symbol table or in the middle of an encoding session to keep a transmitter busy.

A class description for this refined understanding of the requirements on the module is shown in Figure. The class's buffer and current-bit behaviors keep track of the state of the encoding,and the table attribute provides the current symbol table. The class has three methods as follows:

- **Encode** performs the basic encoding function. It takes in a 1-byte input symbol and returns two values: a boolean showing whether it is returning a full buffer and, if the boolean is true, the full buffer itself.





- **New-symbol-table** installs a new symbol table into the object and throws away the current contents of the internal buffer.
- **Flush** returns the current state of the buffer, including the number of valid bits in the buffer.

We also need to define classes for the data buffer and the symbol table. These classes are shown in Figure 3.22.The data-buffer will be used to hold both packed symbols and unpacked ones (such as in the symbol table). It defines the buffer itself and the length of the buffer. We have to define a data type because the longest encoded symbol is longer than an input symbol. The longest Huffman code for an eight-bit input symbol is 256 bits. (Ending up with a symbol this long happens only
when the symbol probabilities have the proper values.) The insert function packs
a new symbol into the upper bits of the buffer; it also puts the remaining bits in a new buffer if the current buffer is overflowed. The Symbol-table class indexes the encoded version of each symbol. The class defines an access behavior for the table; it also defines a load behaviour to create a new symbol table.The relationships between these classes are shown in Figure 3.23—a data compressor object includes one buffer and one symbol table

FIGURE 3.23
Relationships between classes in the data compressor.

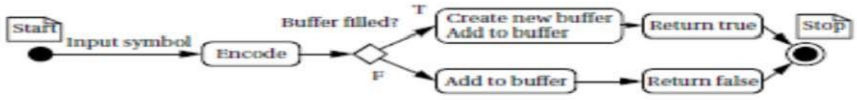
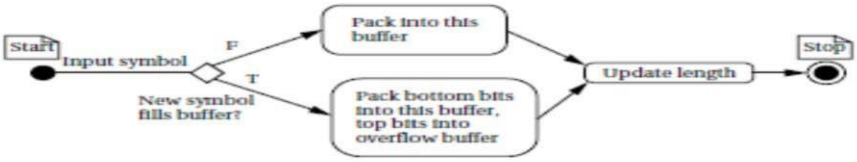


FIGURE 3.24
State diagram for encode behavior.



Program Design

OO design in C

How would we have to modify the implementation for C?We have two choices in implementation, based on whether we want to support multiple simultaneous data compressors. If we want to strictly adhere to the specification,we must be able to run several simultaneous compressors,since in the object-oriented specification we can create as many new data-compressor objects as we want.

We may not have the luxury of coding the algorithm in C++. While C is almost
universally supported on embedded processors,support for languages that support
object orientation such as C++ or Java is not so universal. How would we have to structure C code to provide multiple instantiations of the data compressor? The
fundamental point is that we cannot rely on any global variables—all of the object state must be replicable.We can do this relatively easily,making the code only a little more
Cumber some. We create a structure that holds the data part of the object as follows:

```
struct data_compressor_struct {
data_buffer buffer;
```

```
int current_bit;
sym_table table;
}
typedef struct data_compressor_struct data_compressor,
*data_compressor_ptr; /* data type declaration for
convenience */
```

We would of course have to do something similar for the other classes. Depending on how strict we want to be,we may want to define data access functions to get to fields in the various structures we create. C would permit us to get to those struck fields without using the access functions but using the access functions would give us a little extra freedom to modify the structure definitions later.
We then implement the class methods as C functions, passing in a pointer to the data_compressor object we want to operate on. Appearing below is the beginning of the modified encode method showing how we make explicit all references to the data in the object.

```
typedef char boolean; /* for clarity */
#define TRUE 1
#define FALSE 0
boolean data_compressor_encode(data_compressor_ptr mycmprs,
char isymbol, data_buffer *fullbuf) {
data_buffer temp;
int len, overlen;
/* look up the new symbol */
temp = mycmprs->table[isymbol].value; /* the symbol
itself */
len = mycmprs->table[isymbol].length; /* its value */
...
```

(For C++ afficionados, the above amounts to making explicit the C++ this pointer.)

If, on the other hand, we did not care about the ability to run multiple compressions simultaneously,we can make the functions a little more readable by using global variables for the class variables:

```
static data_buffer buffer;
static int current_bit;
static sym_table table;
```

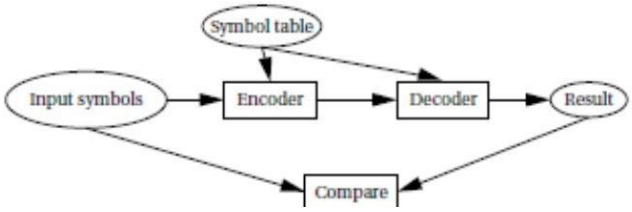
We have used the C static declaration to ensure that these globals are not defined outside the file in which they are defined;this gives us a little added modularity.We would, of course, have to update the specification so that it makes clear that only one compressor object can be running at a time.The functions that implement the methods can then operate directly on the globals as seen below.

```
boolean data_compressor_encode(char isymbol, data_buffer*
fullbuf) {
data_buffer temp;
int len, overlen;
/* look up the new symbol */
temp = table[isymbol].value; /* the symbol itself */
len = table[isymbol].length; /* its value */
```

...
Notice that this code does not need the structure pointer argument, making it resemble the C++ code a little more closely. However, horrible bugs will ensue if we try to run two different compressions at the same time through this code. What can we say about the efficiency of this code? Efficiency has many aspects covered in more detail in Chapter 5. For the moment, let's consider instruction
selection, that is, how well the compiler does in choosing the right instructions to implement the operations. Bit manipulations such as we do here often raise concerns about efficiency. But if we have a good compiler and we select the right data types,instruction selection is usually not a problem. If we use data types that do not require data type transformations, a good compiler can select the right instructions to efficiently implement the required operations.

Testing

In the meantime, we can use common sense to come up with some testing techniques. One way to test the code is to run it and look at the output without considering how the code is written. In this case, we can load up a symbol table, run some symbols through it,and see whether we get the correct result.We can get the symbol table from outside sources



Another way to test the code is to examine the code itself and try to identify potential problem areas. When we read the code,we should look for places where data operations take place to see that they are performed properly.We also want to look at the conditionals to identify different cases that need to be exercised. Some ideas of things to look out for are listed below.

- Is it possible to run past the end of the symbol table?
- What happens when the next symbol does not fill up the buffer?
- What happens when the next symbol exactly fills up the buffer?
- What happens when the next symbol overflows the buffer?
- Do very long encoded symbols work properly? How about very short ones?
- Does flush ( ) work properly?

Testing the internals of code often requires building **scaffolding code**.

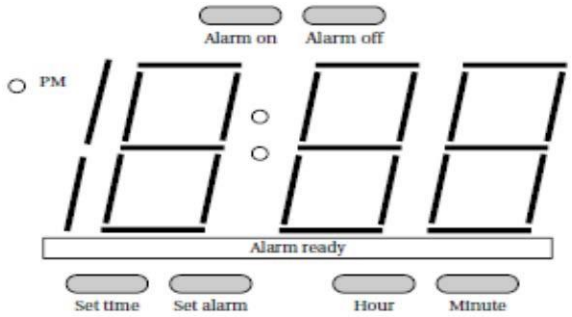
2. Design an Alarm clock. (16) [CO5-H3]

We use a microprocessor to read the clock's buttons and update the time display. Since we now have an understanding of I/O we work through the steps of the methodology to go from a concept to a completed and tested system.

Requirements

The basic functions of an alarm clock are well understood and easy to enumerate. Figure illustrates the front panel design for the alarm clock.The time is shown as four

digits in 12-h format; we use a light to distinguish between AM and PM. We use several buttons to set the clock time and alarm time. When we press the hour and minute buttons,we advance the hour and minute, respectively, by one.
When setting the time, we must hold down the set time button while we hit the hour and minute buttons; the set alarm button works in a similar fashion.We turn the alarm on and off with the alarm on and alarm off buttons. When the alarm is activated, the alarm ready light is on. A separate speaker provides the audiblealarm.

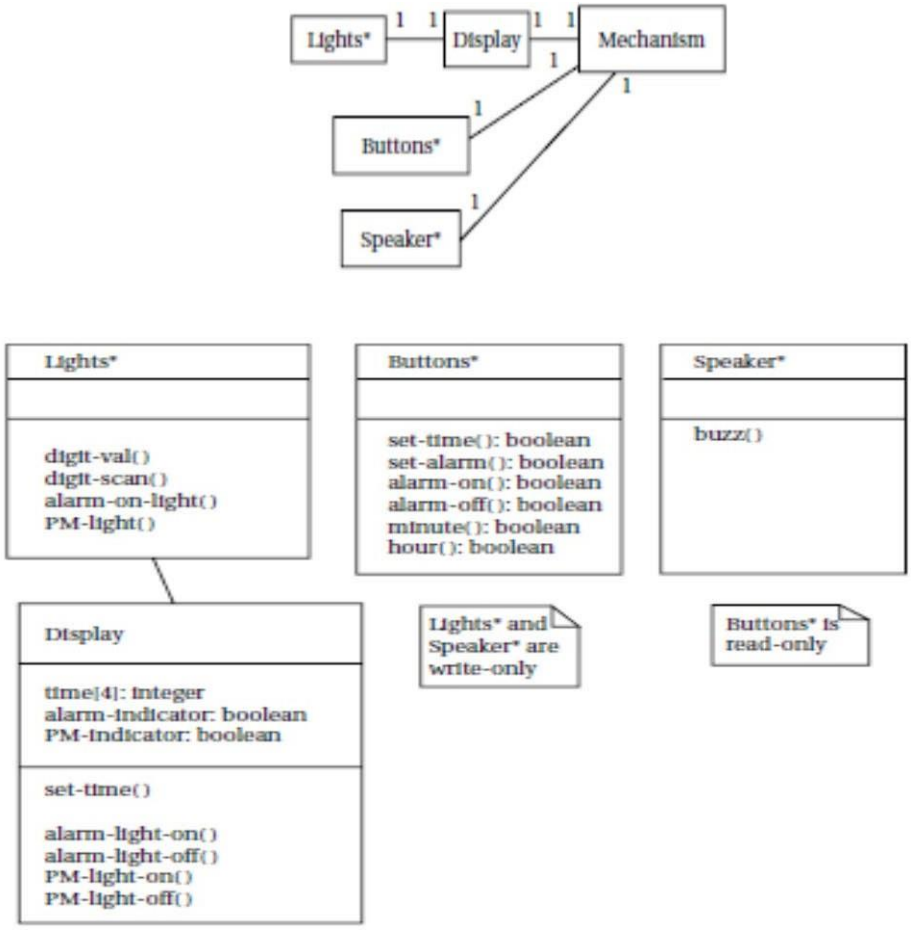


We are now ready to create the requirements table.

Name	Alarm clock.
Purpose	A 24-h digital clock with a single alarm.
Inputs	Six push buttons: set time, set alarm, hour, minute, alarm on, alarm off.
Outputs	Four-digit, clock-style output. PM indicator light. Alarm ready light. Buzzer.
Functions	Default mode: The display shows the current time. PM light is on from noon to midnight. Hour and minute buttons are used to advance time and alarm, respectively. Pressing one of these buttons increments the hour/minute once. Depress set time button: This button is held down while hour/minute buttons are pressed to set time. New time is automatically shown on display. Depress set alarm button: While this button is held down, display shifts to current alarm setting; depressing hour/minute buttons sets alarm value in a manner similar to setting time. Alarm on: puts clock in alarm-on state, causes clock to turn on buzzer when current time reaches alarm time, turns on alarm ready light. Alarm off: turns off buzzer, takes clock out of alarm-on state, turns off alarm ready light.
Performance	Displays hours and minutes but not seconds. Should be accurate within the accuracy of a typical microprocessor clock signal. (Excessive accuracy may unreasonably drive up the cost of generating an accurate clock.)
Manufacturing cost	Consumer product range. Cost will be dominated by the microprocessor system, not the buttons or display.
Power	Powered by AC through a standard power supply.
Physical size and weight	Small enough to fit on a nightstand with expected weight for an alarm clock.

Specification

The basic function of the clock is simple,but we do need to create some classes and associated behaviors to clarify exactly how the user interface works. Figure shows the basic classes for the alarm clock. Borrowing a term from mechanical watches, we call the class that handles the basic clock operation the Mechanism class.We have three classes that represent physical elements: Lights*
for all the digits and lights, Buttons* for all the buttons, and Speaker* for the sound output. The Buttons* class can easily be used directly by Mechanism. As discussed below, the physical display must be scanned to generate the digits output, so we introduce the Display class to abstract the physical lights. The details of the low-level user interface classes are shown in Figure . The
Buzzer* class allows the buzzer to be turned off; we will use analog electronics to generate the buzz tone for the speaker. The Buttons* class provides read-only access to the current state of the buttons. The Lights* class allows us to drive the lights. However, to save pins on the display, Lights* provides signals for only one digit,along with a set of signals to indicate which digit is currently being addressed.



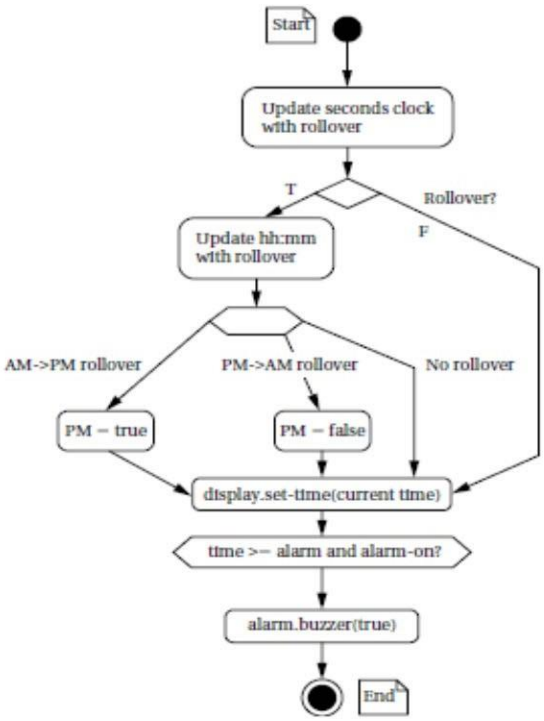
We generate the display by scanning the digits periodically. That function is performed by the Display class, which makes the display appear as an unscanned, continuous display to the rest of the system.
The Mechanism class is described in Figure . This class keeps track of the current time, the current alarm time, whether the alarm has been turned on, and whether it is

currently buzzing. The clock shows the time only to the minute, but it keeps internal time to the second. The time is kept as discrete digits rather than a single integer to simplify transferring the time to the display. The class provides two behaviors, both of which run continuously. First, scan-keyboard is responsible
for looking at the inputs and updating the alarm and other functions as requested by the user. Second, update-time keeps the current time accurate. Figure shows the state diagram for update-time. This behavior is straightforward, but it must do several things. It is activated once per second and must update the seconds clock. If it has counted 60 s, it must then update the displayed
time; when it does so, it must roll over between digits and keep track of AM-to-PM and PM-to-AM transitions. It sends the updated time to the display object. It also compares the time with the alarm setting and sets the alarm buzzing under proper conditions. The state diagram for scan-keyboard is shown in Figure . This function is called periodically frequently enough so that all the user's button presses are caught by the system. Because the keyboard will be scanned several times per second,
we do not want to register the same button press several times. If, for example, we advanced the minutes count on every keyboard scan when the set-time and minutes buttons were pressed the time would be advanced much too fast. To make the buttons respond more reasonably the function computes button activations—it compares the current state of the button to the button's value on the last scan, and it considers the button activated only when it is on for this scan but was off for the
last scan. Once computing the activation values for all the buttons, it looks at the activation combinations and takes the appropriate actions. Before exiting, it saves the current button values for computing activations the next time this behaviour is executed.

System Architecture

The software and hardware architectures of a system are always hard to completely separate, but let's first consider the software architecture and then its implications on the hardware. The system has both periodic and aperiodic components—the current time must obviously be updated periodically, and the button commands occur occasionally. It seems reasonable to have the following two major software components:

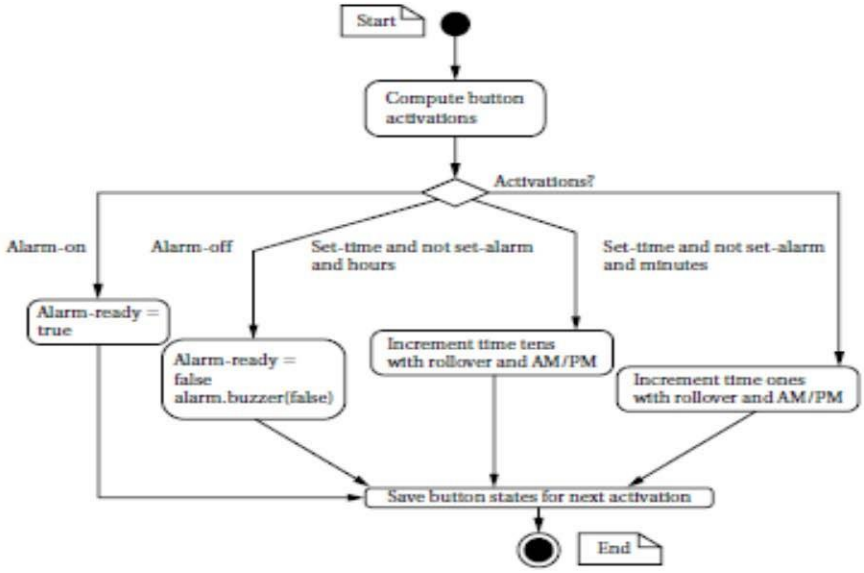
- An interrupt-driven routine can update the current time.The current time will be kept in a variable in memory.A timer can be used to interrupt periodically and update the time. As seen in the subsequent discussion of the hardware



architecture, the display must be sent the new value when the minute value changes. This routine can also maintain the PM indicator.

- A foreground program can poll the buttons and execute their commands.

Since buttons are changed at a relatively slow rate, it makes no sense to add the hardware required to connect the buttons to interrupts. Instead, the foreground program will read the button values and then use simple conditional tests to implement the commands, including setting the current time, setting the alarm and turning off the alarm. Another routine called by the foreground program will turn the buzzer on and off based on the alarm time.

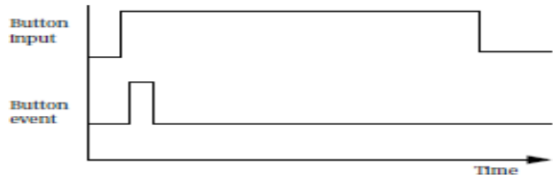


An important question for the interrupt-driven current time handler is how often the timer interrupts occur. A 1-min interval would be very convenient for the software, but a one-minute timer would require a large number of counter bits. It is more realistic to use a one-second timer and to use a program variable to count the seconds in a minute. The foreground code will be implemented as a while loop:

```
while (TRUE) {
read_buttons(button_values);/* read inputs */
process_command(button_values);/* do commands */
check_alarm();/* decide whether to turn on the alarm */
}
```

The loop first reads the buttons using read_buttons(). In addition to reading the current button values from the input device, this routine must pre-process the button values so that the user interface code will respond properly. The buttons will remain depressed for many sample periods since the sample rate is much faster than any person can push and release buttons. We want to make sure that the clock responds to this as a single depression of the button, not one depression per sample
interval. As shown in Figure 4.40, this can be done by performing a simple edge detection on the button input—the button event value is 1 for one sample period when the button is depressed and then goes back to 0 and does not return to 1 until the button is depressed and then released. This can be accomplished by a simple two-state machine.

The process_command() function is responsible for responding to button events. The check_alarm() function checks the current time against the alarm time and decides when to turn on the buzzer. This routine is kept separate from the command processing code since the alarm must go on when the proper time is reached, independent of the button inputs. We have determined from the software architecture that we will need a timer connected to the CPU. We will also need logic to connect the buttons to the CPU bus. In addition to performing edge detection on the button inputs, we must also
of course debounce the buttons. The final step before starting to write code and build hardware is to draw the state transition graph for the clock's commands. That diagram will be used to guide
the implementation of the software components.



Component Design and Testing

The two major software components, the interrupt handler and the foreground code, can be implemented relatively straightforwardly. Since most of the functionality of the interrupt handler is in the interruption process itself, that code is best tested on the microprocessor platform. The foreground code can be more easily tested on the PC or workstation used for code development. We can create a test bench for this code that generates button depressions to exercise the state machine. We will also need to simulate the advancement of the system clock. Trying to directly execute the interrupt

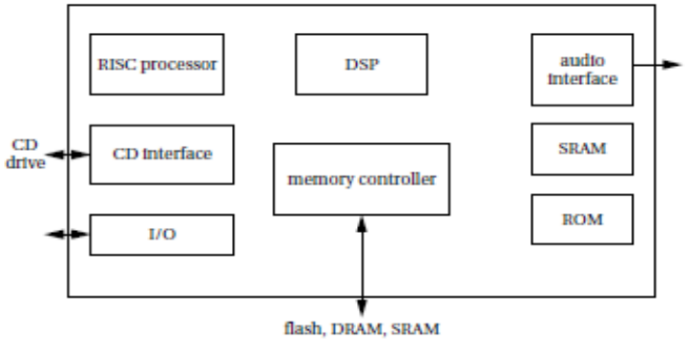
handler to control the clock is probably a bad idea—not only would that require some type of emulation of interrupts, but it would require us to count interrupts second by second. A better testing strategy is to add testing code that updates the clock, perhaps once per four iterations of the foreground while loop. The timer will probably be a stock component, so we would then focus on implementing logic to interface to the buttons, display and buzzer. The buttons will require debouncing logic. The display will require a register to hold the current display value in order to drive the display elements.

System Integration and Testing

Because this system has a small number of components, system integration is relatively easy. The software must be checked to ensure that debugging code has been turned off. Three types of tests can be performed. First, the clock's accuracy can be checked against a reference clock. Second, the commands can be exercised from the buttons. Finally, the buzzer's functionality should be
verified.

3. Design an audio player. (8) [CO5-H3]

Audio players are often called **MP3 players** after the popular audio data format. The earliest portable MP3 players were based on compact disc mechanisms. Modern MP3 players use either flash memory or disk drives to store music. An MP3 player performs three basic functions: audio storage, audio decompression, and user interface. Although audio compression is computationally intensive, audio decompression is relatively lightweight. The incoming bit stream has been encoded using a Huffman-style code, which must be decoded. The audio data itself is applied to a reconstruction filter, along with a few other parameters. MP3 decoding can, for example, be executed using only 10% of an ARM7 CPU. The user interface of an MP3 player is usually kept simple to minimize both the
physical size and power consumption of the device. Many players provide only a simple display and a few buttons.



The file system of the player generally must be compatible with PCs. CD/MP3 players used compact discs that had been created on PCs. Today's players can be plugged into USB ports and treated as disk drives on the host processor. The Cirrus CS7410 [Cir04B] is an audio controller designed for CD/MP3 players. The audio controller includes two processors. The 32-bit RISC processor is
used to perform system control and audio decoding. The 16-bit DSP is used to perform audio effects such as equalization. The memory controller can be interfaced to several different types of memory: flash memory can be used for data or code storage; DRAM

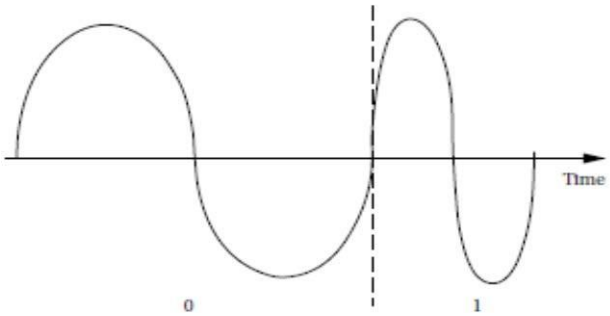
can be used as a buffer to handle temporary disruptions of the CD data stream. The audio interface unit puts out audio in formats that
can be used by A/D converters. General-purpose I/O pins can be used to decode buttons, run displays, etc. Cirrus provides a reference design for a CD/MP3 player [Cir04A].

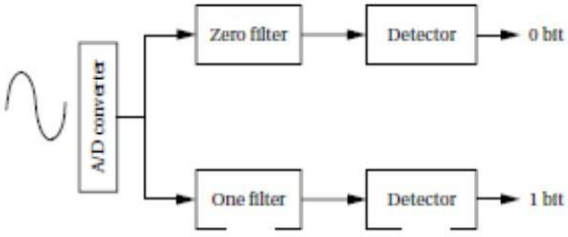
4. Design a software modem. (16) [CO5-H3]

In this section we design a modem. Low-cost modems generally use specialized chips, but some PCs implement the modem functions in software. Before jumping into the modem design itself,we discuss principles of how to transmit digital data over a telephone line. We will then go through a specification and discuss architecture, module design, and testing.

Theory of Operation and Requirements

The modem will use **frequency-shift keying (FSK)**,a technique used in 1200-baud modems. Keying alludes to Morse code—style keying.As shown in Figure , the FSK scheme transmits sinusoidal tones, with 0 and 1 assigned to different frequencies. Sinusoidal tones are much better suited to transmission over analog phone lines than are the traditional high and low voltages of digital circuits. The 01 bit patterns create the chirping sound characteristic of modems. (Higher-speed modems





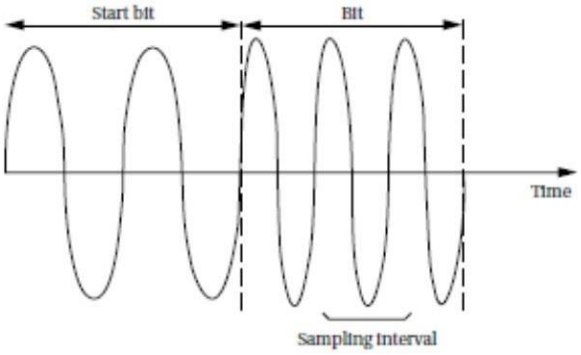
are backward compatible with the 1200-baud FSK scheme and begin a transmission with a protocol to determine which speed and protocol should be used.) The scheme used to translate the audio input into a bit stream is illustrated in Figure .The analog input is sampled and the resulting stream is sent to two digital filters (such as an FIR filter). One filter passes frequencies in the range that represents a 0 and rejects the 1-band frequencies, and the other filter does the converse. The

outputs of the filters are sent to detectors, which compute the average value of the signal over the past n samples. When the energy goes above a threshold value, the appropriate bit is detected.

We will send data in units of 8-bit bytes.The transmitting and receiving modems agree in advance on the length of time during which a bit will be transmitted (otherwise known as the baud rate). But the transmitter and receiver are physically separated and therefore are not synchronized in any way. The receiving modem does not know when the transmitter has started to send a byte. Furthermore, even

when the receiver does detect a transmission, the clock rates of the transmitter and receiver may vary somewhat,causing them to fall out of sync. In both cases,we can reduce the chances for error by sending the waveforms for a longer time. The receiving process is illustrated in Figure . The receiver will detect the start of a byte by looking for a start bit,which is always 0. By measuring the length of

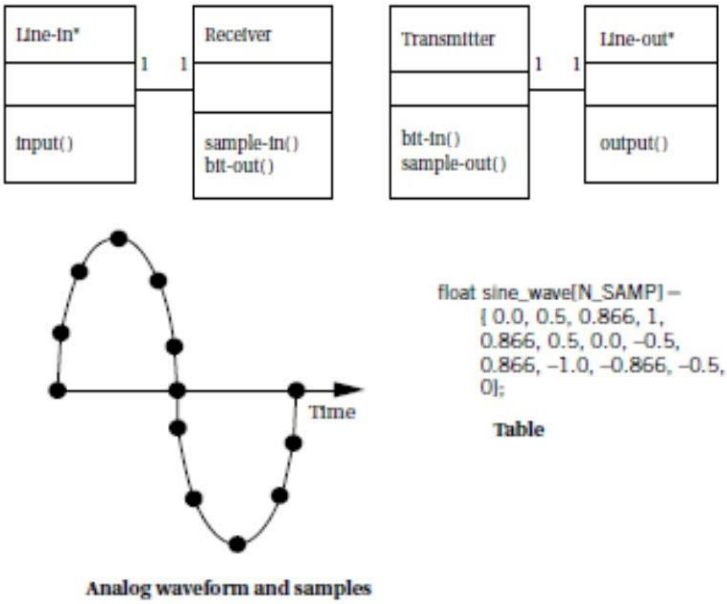
the start bit,the receiver knows where to look for the start of the first bit. However, since the receiver may have slightly misjudged the start of the bit, it does not immediately try to detect the bit. Instead, it runs the detection algorithm at the predicted middle of the bit. The modem will not implement a hardware interface to a telephone line or software for dialing a phone number. We will assume that we have analog audio inputs and outputs for sending and receiving.We will also run at a much slower bit rate than 1200 baud to simplify the implementation. Next,we will not implement a serial interface to a host, but rather put the transmitter's message in memory and save the receiver's result in memory as well. Given those understandings, let's fill out the requirements table.



Name	Modem.
Purpose	A fixed baud rate frequency-shift keyed modem.
Inputs	Analog sound input, reset button.
Outputs	Analog sound output, LED bit display.
Functions	Transmitter: Sends data stored in microprocessor memory in 8-bit bytes. Sends start bit for each byte equal in length to one bit. Receiver: Automatically detects bytes and stores results in main memory. Displays currently received bit on LED.
Performance	1200 baud.
Manufacturing cost	Dominated by microprocessor and analog I/O.
Power	Powered by AC through a standard power supply.
Physical size and weight	Small and light enough to fit on a desktop.

System Architecture

The modem consists of one small subsystem (the interrupt handlers for the samples) and two major subsystems (transmitter and receiver).Two sample interrupt handlers are required, one for input and another for output, but they are very simple. The transmitter is simpler, so let's consider its software architecture first. The best way to generate waveforms that retain the proper shape over long intervals is **table lookup**. Software oscillators can be used to generate periodic signals, but numerical problems limit their accuracy. Figure shows an analog waveform with sample points and the C code for these samples. Table lookup can be combined with interpolation to generate high-resolution waveforms without excessive memory costs, which is more accurate than oscillators because no feedback is involved. The required number of samples for the modem can be found by
experimentation with the analog/digital converter and the sampling code. The structure of the receiver is considerably more complex. The filters and detectors of Figure can be implemented with circular buffers. But that module must feed a state machine that recognizes the bits. The recognizer state machine must use a timer to determine when to start and stop computing the filter output average
based on the starting point of the bit. It must then determine the nature of the bit at the proper interval. It must also detect the start bit and measure it using the counter. The receiver sample interrupt handler is a natural candidate to double as the receiver timer since the receiver's time points are relative to samples. The hardware architecture is relatively simple. In addition to the analog/digital
and digital/analog converters, a timer is required. The amount of memory required to implement the algorithms is relatively small.



Analog waveform and samples

Component Design and Testing:

The transmitter and receiver can be tested relatively thoroughly on the host platform since the timing-critical code only delivers data samples. The transmitter's output is relatively easy to verify, particularly if the data are plotted. A test bench can be constructed to feed the receiver code sinusoidal inputs and test its bit recognition rate. It

is a good idea to test the bit detectors first before testing the complete receiver operation. One potential problem in host-based testing of the receiver is
encountered when library code is used for the receiver function. If a DSP library for the target processor is used to implement the filters, then a substitute must be found or built for the host processor testing. The receiver must then be retested when moved to the target system to ensure that it still functions properly with the library code. Care must be taken to ensure that the receiver does not run too long and miss its deadline. Since the bulk of the computation is in the filters, it is relatively simple to estimate the total computation time early in the implementation process.

System Integration and Testing:

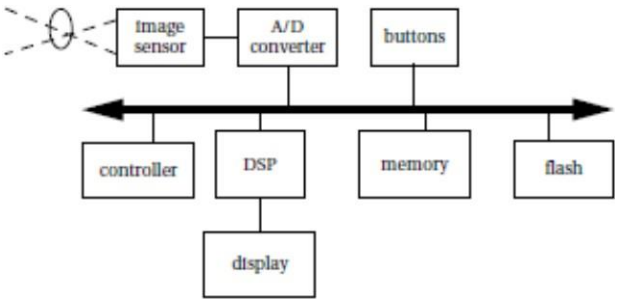
There are two ways to test the modem system: by having the modem's transmitter send bits to its receiver, and or by connecting two different modems. The ultimate test is to connect two different modems particularly modems designed by different people to be sure that incompatible assumptions or errors were not made. But single-unit testing, called **loop-back** testing in the telecommunications industry, is simpler and a good first step. Loop-back can be performed in two ways. First, a shared variable can be used to directly pass data from the transmitter to the receiver. Second, an audio cable can be used to plug the analog output to the analog input. In this case it is also possible to inject analog noise to test the resiliency of the detection algorithm.

5. **Design a digital still camera. (16) [CO5-H3]**

The digital still camera bears some resemblance to the film camera but is fundamentally different in many respects. The digital still camera not only captures images, it also performs a substantial amount of image processing that formerly was done by photofinishers. Digital image processing allows us to fundamentally rethink the camera. A simple example is digital zoom, which is used to extend or replace optical zoom. Many cell phones include digital cameras,creating a hybrid imaging/communication device.

Digital still cameras must perform many functions:

- It must determine the proper exposure for the photo.
- It must display a preview of the picture for framing.
- It must capture the image from the image sensor.
- It must transform the image into usable form.
- It must convert the image into a usable format, such as JPEG, and store the image in a file system.



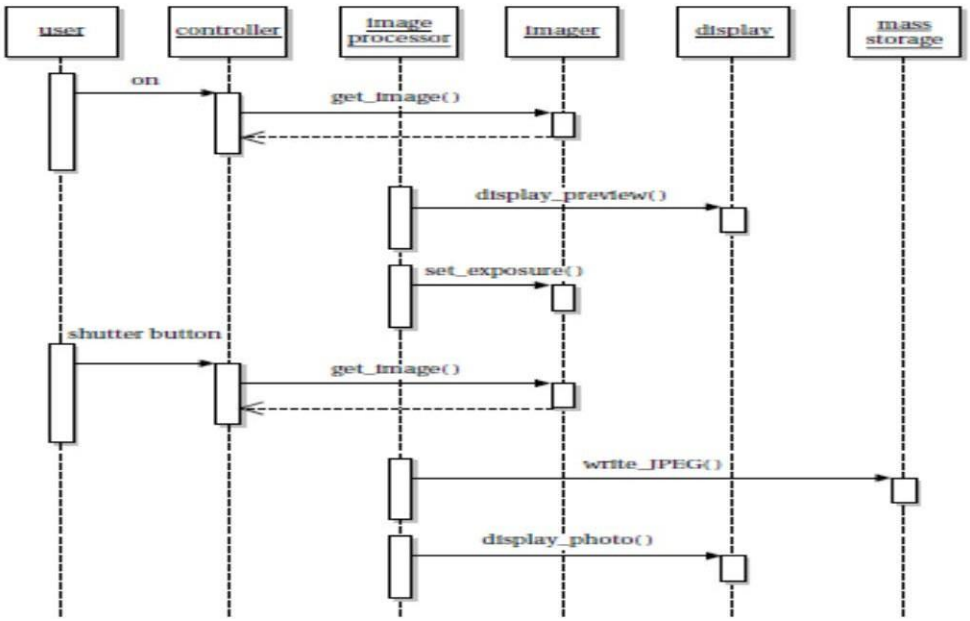
A typical hardware architecture for a digital still camera is shown in Figure . Most cameras use two processors. The controller sequences operations on the camera and performs operations like file system management. The DSP concentrates on image processing. The DSP may be either a programmable processor or a set of hardwired accelerators. Accelerators are often used to minimize power consumption.

The picture taking process can be divided into three main phases: composition, capture, and storage. We can better understand the variety of functions that must be performed by the camera through a sequence diagram. Figure shows a sequence diagram for taking a picture using a point-and-shoot digital still camera. As we walk through this sequence diagram, we can introduce some concepts in digital photography.

When the camera is turned on, it must start to display the image on the camera's screen. That imagery comes from the camera's image sensor. To provide a reasonable image, it must adjust the image exposure. The camera mechanism provides two basic exposure controls: shutter speed and aperture. The camera also displays what is seen through the lens on the camera's display. In general, the display has fewer pixels than does the image sensor; the image processor must generate a smaller version of the image. When the user depresses the shutter button, a number of steps occur. Before the image is captured, the final exposure must be determined. Exposure is computed by analyzing the image characteristics; histograms of the distribution of pixel brightness are often used to determine focus. The camera must also determine **white balance**. Different sources of light, such as sunlight and incandescent lamps, provide light of different colors. The eye naturally compensates for the color of incident light; the camera must perform comparable processing to avoid giving the picture a color cast.

White balance algorithms generally use color histograms to determine the range of colors and re-weigh colors to reduce casts. The image captured from the image sensor is not directly usable, even after exposure and white balance. Virtually all still cameras use a single image sensor to capture a color image.

Color is captured using microscopic color filters, each the size of a pixel, over the image sensor. Since each pixel can capture only one color, the color filters must be arranged in a pattern across the image sensor. A commonly used pattern is the Bayer pattern [Bay75] shown in Figure.



This pattern uses two greens for every red and blue pixel since the human eye is most sensitive to green. The camera must interpolate colors so that every pixel has red, green, and blue values.

After this image processing is complete, the image must be compressed and saved. Images are often compressed in JPEG format, but other formats, such as GIF, may also be used. The EXIF standard (http://www.exif.org) defines a file format for data interchange. Standard compressed image formats such as JPEG are components of an EXIF image file; the EXIF file may also contain a thumbnail image for preview, metadata about the picture such as when it was taken, etc.

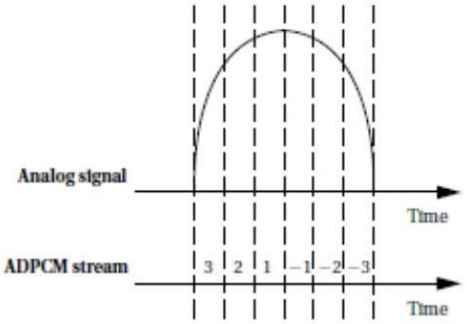
Image compression need not be performed strictly in real time. However, many cameras allow users to take a burst of images ,in which case the images must be compressed quickly to make room in the image processing pipeline for the next image. Buffering is very important in digital still cameras. Image processing often takes longer than capturing an image. Users often want to take a burst of several pictures, for example during sports events. A buffer memory is used to capture the image
from the sensor and store it until it can be processed by the DSP [Sas91]. The display is often connected to the DSP rather than the system bus. Because the display is of lower resolution than the image sensor, the images from the image sensor must be reduced in resolution. Many still cameras use displays originally designed for camcorders, so the DSP may also need to clip the image to accommodate the differing aspect ratios of the display and image sensor.

6. Design a telephone answering machine. (16) [CO5-H3]

In this section we design a digital telephone answering machine. The system will store messages in digital form rather than on an 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.

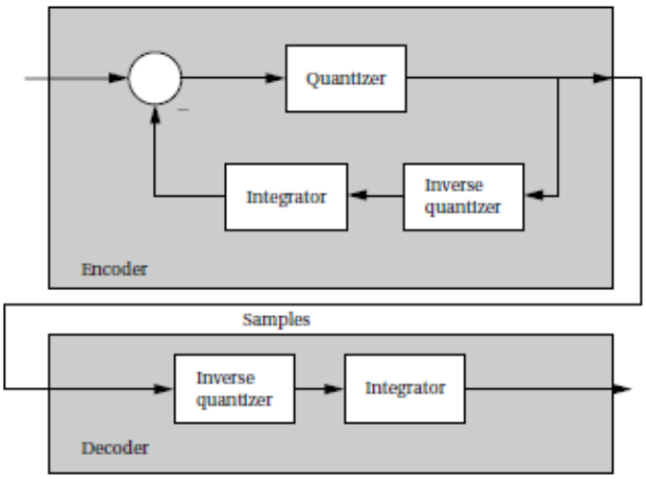
Theory of Operation and Requirements:

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 Figure 6.19. 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. AnADPCM compression system, including an encoder and decoder, is shown in Figure. 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 reconstructs the original signal. The encoder compares the incoming waveform to the predicted waveform (the waveform that will be generated in the decoder). The quantizer encodes this difference as the best predictor of the next waveform value. The inverse quantizer allows us to map bit-level symbols onto real numerical values; for example, the eight possible codes in a 3-bit code can be mapped onto floating-point numbers. The decoder simply uses an inverse quantizer and integrator to turn the differential samples into the waveform.
The answering machine will ultimately be connected to a telephone **subscriber line** (although for testing purposes we will construct a simulated line). At the other end of the subscriber line is the **central office**. All information is carried on the phone line in analog form over a pair of wires. In addition to analog/digital and digital/analog converters to send and receive voice data, we need to sense two other characteristics of the line.



- Ringing: The central office sends a ringing signal to the telephone when a call is waiting. The ringing signal is in fact a 90V 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 **offhook**; 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

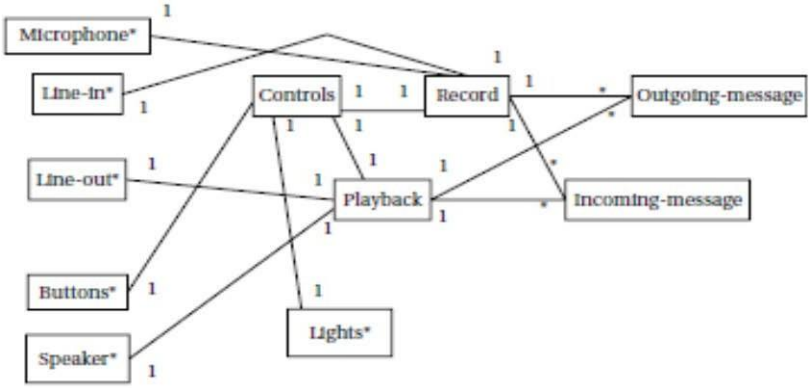
telephone line simulator and then build the analog circuitry necessary to connect to a real phone line.We will use the term **outgoing message**
(OGM) to refer to the message recorded by the owner of the machine and played at the start of every phone call.
We have made a few arbitrary decisions about the user interface in these requirements. The amount of voice data that can be saved by the machine should in fact be determined by two factors: the price per unit of DRAM at the time at which the device goes into manufacturing (since the cost will almost certainly drop from the start of design to manufacture) and the projected retail price at which the machine must sell. The protocol when the memory is full is also arbitrary—it would make at least as much sense to throw out old messages and replace them with new ones, and ideally the user could select which protocol to use. Extra features such as an indicator showing the number of messages or a save messages feature would also be nice to have in a real consumer product.

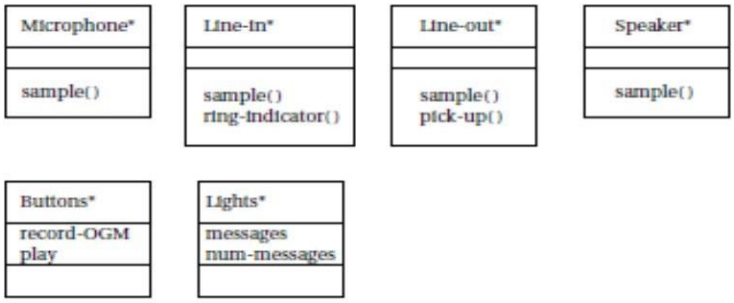
Name	Digital telephone answering machine
Purpose	Telephone answering machine with digital memory, using speech compression.
Inputs	*Telephone:* voice samples, ring indicator. *User interface:* microphone, play messages button, record OGM button.
Outputs	*Telephone:* voice samples, on-hook/off-hook command. *User interface:* speaker, # messages indicator, message light.
Functions	*Default mode:* When machine receives ring indicator, it signals off-hook, plays the OGM, and then records the incoming message. Maximum recording length for incoming message is 30 s, at which time the machine hangs up. If the machine runs out of memory, the OGM is played and the machine then hangs up without recording. *Playback mode:* When the play button is depressed, the machine plays all messages. If the play button is depressed again within five seconds, the messages are played again. Messages are erased after playback. *OGM editing mode:* When the user hits the record OGM button, the machine records an OGM of up to 10 s. When the user holds down the record OGM button and hits the play button, the OGM is played 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 8 kHz.
Manufacturing cost	Consumer product range: approximately $50.
Power	Powered by AC through a standard power supply.
Physical size and weight	Comparable in size and weight to a desk telephone.

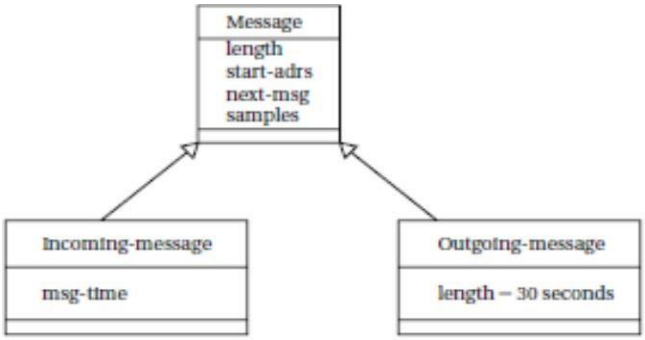
Specification:

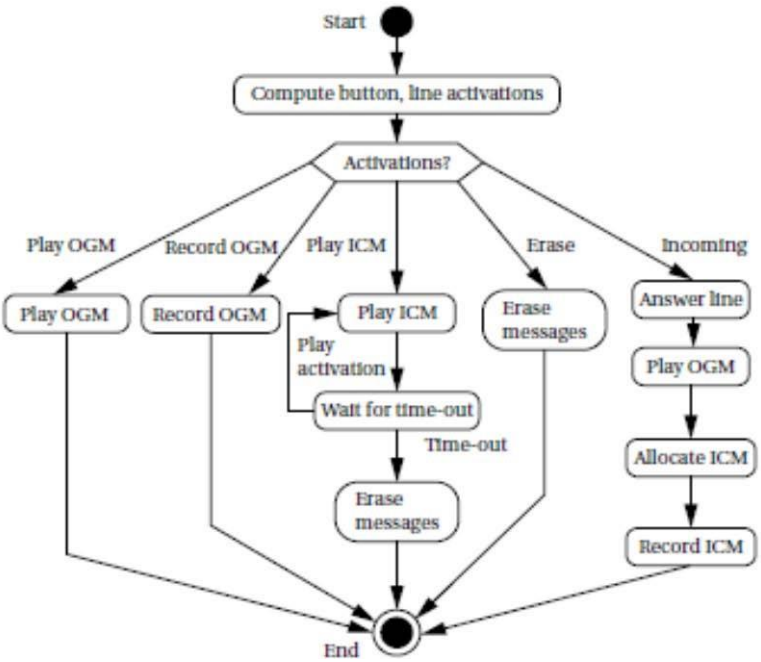
Figure1 shows the class diagram for the answering machine. In addition to the classes that perform the major functions, we also use classes to describe the incoming and OGMs. As seen below, these classes are related. The definitions of the physical interface classes are shown in Figure2. The buttons and lights simply provide attributes for their input and output values. The phone line, microphone, and speaker are given behaviors that let us sample their current values. The message classes are defined in Figure 3. 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 Figure4. The Controls class provides an operate(

) behavior that oversees the user-level operations. 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 Figure5. Most of the user activities are relatively straightforward. The most complex is answering an incoming call. As with the software modem of Section 5.11,we 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. State diagrams for record-msg and playback-msg are shown in Figure6. We have parameterized the specification for record-msg so that it can be used either from the phone line or from the microphone. This requires parameterizing the source itself and the termination condition.







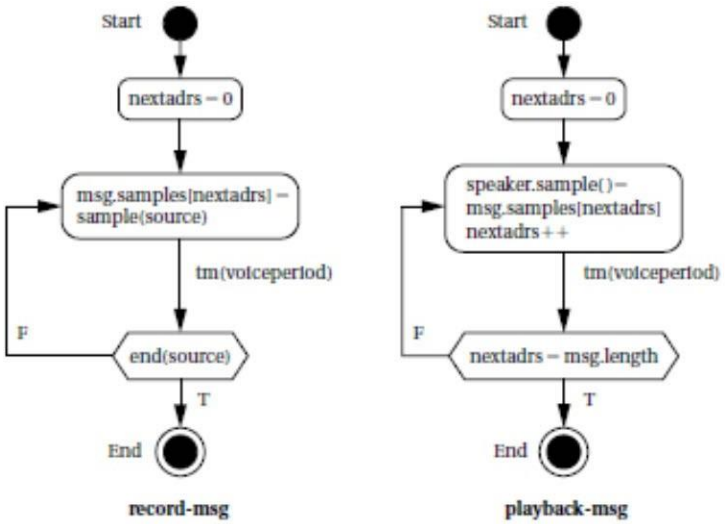


System Architecture:

The machine consists of two major subsystems from the user's point of view: the user interface and the telephone interface. The user and telephone interfaces both appear internally as I/O devices on the CPU bus with the main memory serving as the storage for the messages.

The software splits into the following seven major pieces:

- The **front panel module** handles the buttons and lights.
- The **speaker module** handles sending data to the user's speaker.
- The **telephone line module** handles off-hook detection and 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 processing. These modules don't run all the time and so can be disabled by the front panel and telephone line modules when they are not needed.

▪ The compression and decompression modules run at the same rate as the speaker and telephone I/O modules, but they are not directly connected to devices.We will therefore call them as subroutines to the interrupt modules.

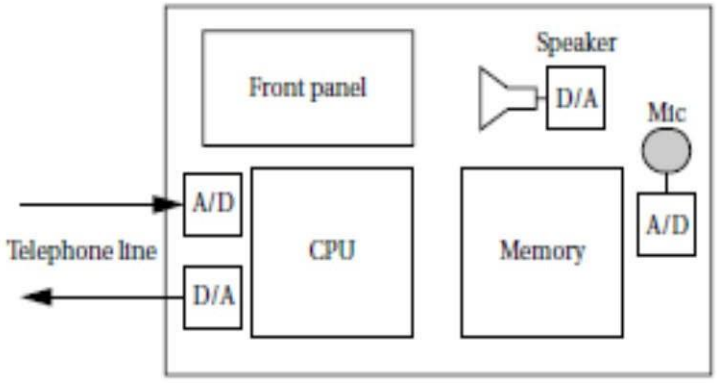
One subtlety is that we must construct a very simple file system for messages, since we have a variable number of messages of variable lengths. Since messages vary in length, we must record the length of each one. In this simple specification, because we always play back the messages in the order in which they were recorded, we don't have to keep a full-fledged directory. If we allowed users to selectively delete messages and save others, we would have to build some sort of directory structure for the messages.

The hardware architecture is straightforward and illustrated in Figure. The speaker and telephone I/O devices appear as standard A/D and D/A converters. The telephone line appears as a one-bit input device (ring detect) and a one bit output device (off-hook/on-hook). The compressed data are kept in main memory.

Component Design and Testing:

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.



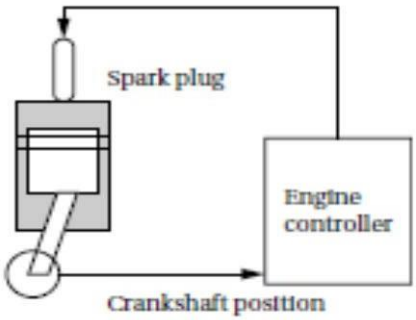
System Integration and Testing:

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 damage 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. The phone line simulator consists of A/D and D/A converters plus a speaker and microphone for voice data, an LED for off-hook/on-hook indication, and a button for ring generation. The telephone
line interface can easily be adapted to connect to these components, and for purposes of testing the answering machine the simulator behaves identically to the real phone line.

7. Design an Automotive engine control. (12) [CO5-H3]

The simplest automotive engine controllers, such as the ignition controller for a basic motorcycle engine, perform only one task—timing the firing of the spark plug, which takes the place of a mechanical distributor. The spark plug must be fired at a certain point in the combustion cycle, but to obtain better performance, the phase relationship between the piston's movement and the spark should change as a function of engine speed. Using a microcontroller that senses the engine crankshaft position allows the spark timing to vary with engine speed.

Firing the spark plug is a periodic process (but note that the period depends on the engine's operating speed).



The control algorithm for a modern automobile engine is much more complex, making the need for microprocessors that much greater. Automobile engines must meet strict requirements (mandated by law in the United States) on both emissions and fuel economy. On the other hand, the engines must still satisfy customers not only in terms of performance but also in terms of ease of starting in extreme cold and heat, low maintenance, and so on. Automobile engine controllers use additional sensors, including the gas pedal position and an oxygen sensor used to control emissions. They also use a multimode control scheme. For example, one mode may be used for engine warm-up, another for cruise, and yet another for climbing steep hills, and so forth. The larger number of sensors and modes increases the number of discrete tasks that must be performed. The highest-rate task is still firing the spark plugs. The throttle setting must be sampled and acted upon regularly, although not as
frequently as the crankshaft setting and the spark plugs. The oxygen sensor responds much more slowly than the throttle, so adjustments to the fuel/air mixture suggested by the oxygen sensor can be computed at a much lower rate. The engine controller takes a variety of inputs that determine the state of the engine. It then controls two basic engine parameters: the spark plug firings and the fuel/air mixture. The engine control is computed periodically, but the periods of the different inputs and outputs range over several orders of magnitude of time. An early paper on automotive electronics by Marley [Mar78] described the rates at which engine inputs and outputs must be handled.

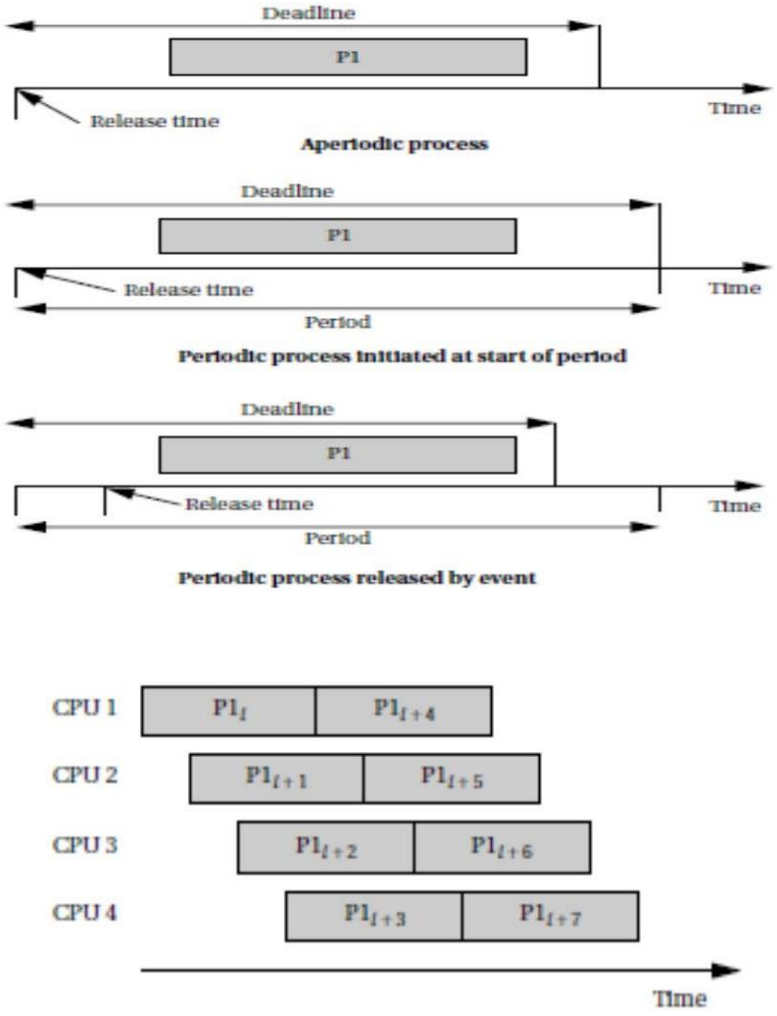
Variable	Time to move full range (ms)	Update period (ms)
Engine spark timing	300	2
Throttle	40	2
Airflow	30	4
Battery voltage	80	4
Fuel flow	250	10
Recycled exhaust gas	500	25
Set of status switches	100	50
Air temperature	seconds	500
Barometric pressure	seconds	1000
Spark/dwell	10	1
Fuel adjustments	80	4
Carburetor adjustments	500	25
Mode actuators	100	100

Timing Requirements on Processes:

Processes can have several different types of timing requirements imposed on them

by the application. The timing requirements on a set of processes strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid. Before studying scheduling proper, we outline the types of process timing requirements that are useful in embedded system design. Figure illustrates different ways in which we can define two important requirements on processes: **release time** and **deadline**. The release time is the time at which the process becomes ready to execute; this is not necessarily the time at which it actually takes control of the CPU and starts to run. An a periodic process is by definition initiated by an event, such as external data arriving or data computed by another process. The release time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself. For a periodically executed process, there are two common possibilities. In simpler systems, the process may become ready at the beginning of the period. More sophisticated systems, such as those with data dependencies between processes, may set the release time at the arrival time of certain data, at a time after the start of the period.
A deadline specifies when a computation must be finished. The deadline for an a periodic process is generally measured from the release time, since that is the only reasonable time reference. The deadline for a periodic process may in general occur at some time other than the end of the period.

Rate requirements are also fairly common. A rate requirement specifies how quickly processes must be initiated. The **period** of a process is the time between successive executions. For example, the period of a digital filter is defined by the time interval between successive input samples. The process's **rate** is the inverse of its period. In a multirate system, each process executes at its own distinct rate. The most common case for periodic processes is for the initiation interval to be equal to
the period. However, pipelined execution of processes allows the initiation interval to be less than the period. Figure illustrates process execution in a system with four CPUs. The various execution instances of program P1 have been subscripted to distinguish their initiation times. In this case, the initiation interval is equal to onefourth of the period. It is possible for a process to have an initiation rate less than the period even in single-CPU systems. If the process execution time is significantly
less than the period, it may be possible to initiate multiple copies of a program at slightly offset times.



8. Design a video accelerator. (16) [CO5-H3]

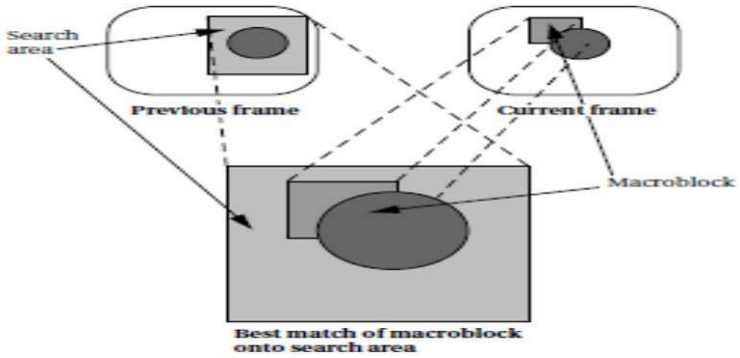
Digital video is still a computationally intensive task, so it is well suited to acceleration. Motion estimation engines are used in real-time search engines; we may want to have one attached to our personal computer to experiment with video processing techniques

Algorithm and Requirements:

We could build an accelerator for any number of digital video algorithms. We will choose **block motion estimation** as our example here because it is very computation and memory intensive but it is relatively easy to explain. Block motion estimation is used in digital video compression algorithms so that one frame in the video can be described in terms of the differences between it and another frame. Because objects in the frame often move relatively little, describing one frame in terms of another greatly reduces the number of bits required to describe the video.

The concept of block motion estimation is illustrated in Figure. The goal is to perform a two-dimensional correlation to find the best match between regions in the two frames. We divide the current frame into **macroblocks** (typically,16_16). For every macro block in the frame, we want to find the region in the previous frame that most closely matches

the macro block. Searching over the entire previous frame would be too expensive, so we usually limit the search to a given area, centered
around the macro block and larger than the macro block. We try the macro block at various offsets in the search area. We measure similarity using the following sum-of-differences measure:



$$\sum_{1 \le i,\, j \le n} \left| M(i,j) - S(i - o_x, j - o_y) \right|,$$

where M(i, j) is the intensity of the macroblock at pixel i, j, S(i, j) is the intensity of the search region,n is the size of the macroblock in one dimension, and ox, oy is the offset between the macroblock and search region. Intensity is measured as an 8-bit luminance that represents a monochrome pixel—color information is not used in motion estimation.We choose the macroblock position relative to the search area that gives us the smallest value for this metric. The offset at this chosen position describes a vector from the search area center to the macroblock's center that is called the **motion vector**. For simplicity, we will build an engine for a full search, which compares the macroblock and search area at every possible point. Because this is an expensive operation, a number of methods have been proposed for conducting a sparser search of the search area. These methods introduce extra control that would cloud our discussion, but these algorithms may provide good examples. A good way to describe the algorithm is in C. Some basic parameters of the algorithm are illustrated in Figure 7.27. Appearing below is the C code for a single search,which assumes that the search region does not extend past the boundary of the frame.

```
bestx = 0; besty = 0; /* initialize best location-none yet */
bestsad = MAXSAD; /* best sum-of-difference thus far */
for (ox = –SEARCHSIZE; ox < SEARCHSIZE; ox++) {
/* x search ordinate */
for (oy = –SEARCHSIZE; oy < SEARCHSIZE; oy++) {
/* y search ordinate */
int result = 0;
for (i = 0; i < MBSIZE; i++) {
for (j = 0; j < MBSIZE; j++) {
result = result + iabs(mb[i][j] – search[i – ox
+ XCENTER][j – oy + YCENTER]);
}
}
```

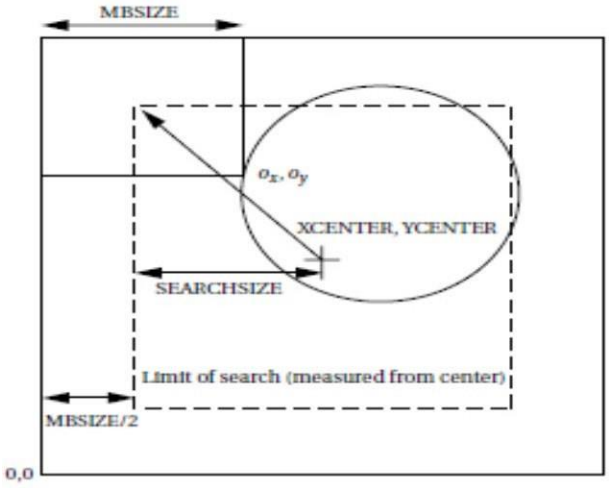
```
if (result <= bestsad) { /* found better match */
bestsad = result;
bestx = ox; besty = oy;
}
}
```

The arithmetic on each pixel is simple, but we have to process a lot of pixels. If MBSIZE is 16 and SEARCHSIZE is 8, and remembering that the search distance in each dimension is 8 _ 1 _ 8, then we must perform

$$N_{OPS} = (16X16)X(17X17) = 73984$$

different operations to find the motion vector for a single macroblock, which requires looking at twice as many pixels, one from the search area and one from the macroblock. (We can now see the interest in algorithms that do not require a full search.) To process video,we will have to perform this computation on every macroblock of every frame. Adjacent blocks have overlapping search areas, so we will try to avoid reloading pixels we already have. One relatively low-resolution standard video format, common intermediate format, has a frame size of 352_288, which gives an array of 22_18 macroblocks. If we want to encode video, we would have to perform motion estimation on every macroblock of most frames (some frames are sent without using motion compensation).

We will build the system using an FPGA connected to the PCI bus of a personal computer. We clearly need a high-bandwidth connection such as the PCI between the accelerator and the CPU. We can use the accelerator to experiment with video processing, among other things. Appearing below are the requirements for the system.



Name	Block motion estimator
Purpose	Perform block motion estimation within a PC system
Inputs	Macroblocks and search areas
Outputs	Motion vectors
Functions	Compute motion vectors using full search algorithms
Performance	As fast as we can get
Manufacturing cost	Hundreds of dollars
Power	Powered by PC power supply
Physical size and weight	Packaged as PCI card for PC

Specification:

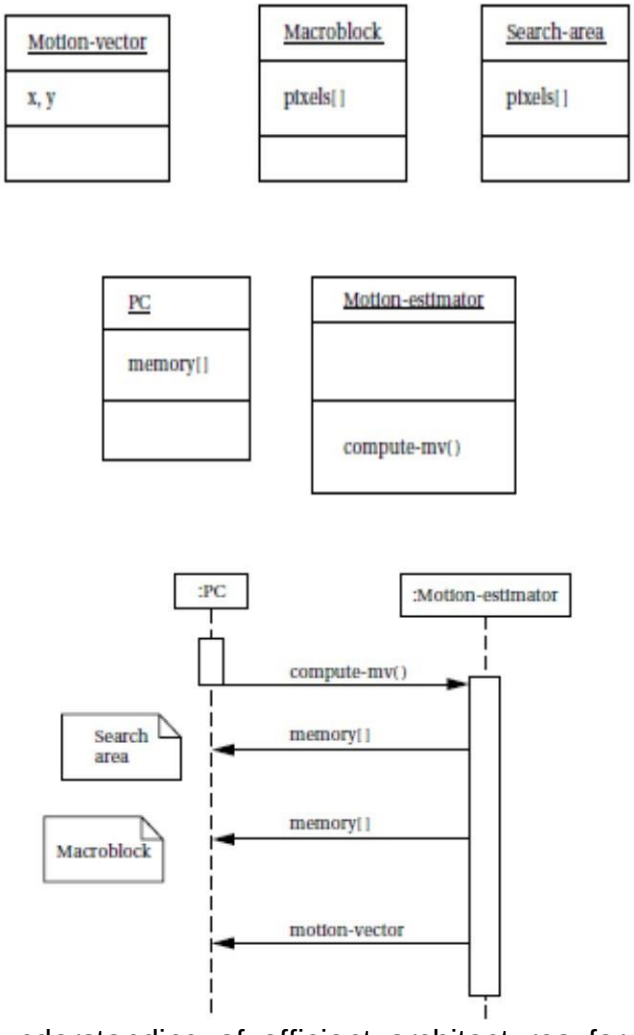
The specification for the system is relatively straightforward because the algorithm is simple. Figure defines some classes that describe basic data types in the system: the motion vector, the macroblock, and the search area. These definitions are straightforward. Because the behavior is simple, we need to define only two classes to describe it: the accelerator itself and the PC. These classes are shown in Figure . The PC makes its memory accessible to the accelerator. The accelerator provides a behavior compute-mv( ) that performs the block motion estimation algorithm. Figure shows a sequence diagram that describes the operation of compute-mv( ). After initiating the behavior, the accelerator reads the search area and macroblock from the PC; after computing the motion vector, it returns it to the PC.

Architecture:

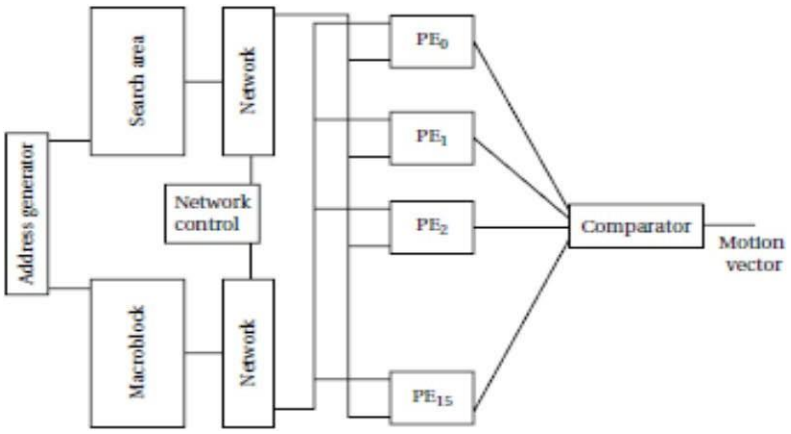
The accelerator will be implemented in an FPGA on a card connected to a PC's PCI slot. Such accelerators can be purchased or they can be designed from scratch. If you design such a card from scratch, you have to decide early on whether the card will be used only for this video accelerator or if it should be made general enough to support other applications as well. The architecture for the accelerator requires some thought because of the large amount of data required by the algorithm. The macroblock has 16_16 _ 256; the search area has (8_8_1_8_8)2 _ 1,089 pixels. The FPGA probably will not have enough memory to hold 1,089 8-bit values.We have to use a memory external to the FPGA but on the accelerator board to hold the pixels.

There are many possible architectures for the motion estimator. One is shown in Figure . The machine has two memories, one for the macroblock and another for the search memories. It has 16 PEs that perform the difference calculation on a pair of pixels; the comparator sums them up and selects the best value to find the motion vector. This architecture can be used to implement algorithms other than a full search by changing the address generation and control. Depending on the number of different motion estimation algorithms that you want to execute on the machine, the networks connecting the memories to the PEs may also be simplified. Figure shows how we can schedule the transfer of pixels from the memories to the PEs in order to efficiently compute a full search on this architecture. The schedule fetches one pixel from the macroblock memory and (in steady state) two pixels from the search area memory per clock cycle. The pixels are distributed to the PEs in a regular pattern as shown by the

schedule.This schedule computes 16 correlations between the macroblock and search area simultaneously. The computations for each correlation are distributed among the PEs; the comparator is responsible for collecting the results, finding the best match value, and remembering the corresponding motion vector.



Based on our understanding of efficient architectures for accelerating motion estimation, we can derive a more detailed definition of the architecture in UML, which is shown in Figure 7.33. The system includes the two memories for pixels, one a single-port memory and the other dual ported. A bus interface module is responsible for communicating with the PCI bus and the rest of the system. The estimation engine reads pixels from the M and S memories, and it takes commands from the bus interface and returns the motion vector to the bus interface.

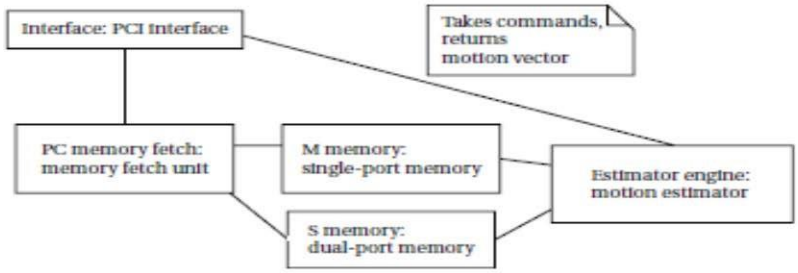


Component Design:

If we want to use a standard FPGA accelerator board to implement the accelerator, we must first make sure that it provides the proper memory required for M and S. Once we have verified that the accelerator board has the required structure,we can concentrate on designing the FPGA logic. Designing an FPGA is, for the most part, a straightforward exercise in logic design. Because the logic for the accelerator is very regular,we can improve the FPGA's clock rate by properly placing the logic in the FPGA to reduce wire lengths. If we are designing our own accelerator board, we have to design both the video accelerator design proper and the interface to the PCI bus.We can create and exercise the video accelerator architecture in a hardware description language like VHDL or Verilog and simulate its operation. Designing the PCI interface requires somewhat different techniques since we may not have a simulation model for a PCI

t	M	S	S9	PE_0	PE_1	PE_2
0	M(0,0)	S(0,0)		|M(0,0) – S(0,0)|		
1	M(0,1)	S(0,1)		|M(0,1) – S(0,1)|	|M(0,0) – S(0,1)|	
2	M(0,2)	S(0,2)		|M(0,2) – S(0,2)|	|M(0,1) – S(0,2)|	|M(0,0) – S(0,2)|
3	M(0,3)	S(0,3)		|M(0,3) – S(0,3)|	|M(0,2) – S(0,3)|	|M(0,1) – S(0,3)|
4	M(0,4)	S(0,4)		|M(0,4) – S(0,4)|	|M(0,3) – S(0,4)|	|M(0,2) – S(0,4)|
5	M(0,5)	S(0,5)		|M(0,5) – S(0,5)|	|M(0,4) – S(0,5)|	|M(0,3) – S(0,5)|
6	M(0,6)	S(0,6)		|M(0,6) – S(0,6)|	|M(0,5) – S(0,6)|	|M(0,4) – S(0,6)|
7	M(0,7)	S(0,7)		|M(0,7) – S(0,7)|	|M(0,6) – S(0,7)|	|M(0,5) – S(0,7)|
8	M(0,8)	S(0,8)		|M(0,8) – S(0,8)|	|M(0,7) – S(0,8)|	|M(0,6) – S(0,8)|
9	M(0,9)	S(0,9)		|M(0,9) – S(0,9)|	|M(0,8) – S(0,9)|	|M(0,7) – S(0,9)|
10	M(0,10)	S(0,10)		|M(0,10) – S(0,10)|	|M(0,9) – S(0,10)|	|M(0,8) – S(0,10)|
11	M(0,11)	S(0,11)		|M(0,11) – S(0,11)|	|M(0,10) – S(0,11)|	|M(0,9) – S(0,11)|
12	M(0,12)	S(0,12)		|M(0,12) – S(0,12)|	|M(0,11) – S(0,12)|	|M(0,10) – S(0,12)|
13	M(0,13)	S(0,13)		|M(0,13) – S(0,13)|	|M(0,12) – S(0,13)|	|M(0,11) – S(0,13)|
14	M(0,14)	S(0,14)		|M(0,14) – S(0,14)|	|M(0,13) – S(0,14)|	|M(0,12) – S(0,14)|
15	M(0,15)	S(0,15)		|M(0,15) – S(0,15)|	|M(0,14) – S(0,15)|	|M(0,13) – S(0,15)|
16	M(1,0)	S(1,0)	S(0,16)	|M(1,0) – S(1,0)|	|M(0,15) – S(0,16)|	|M(0,14) – S(0,16)|
17	M(1,1)	S(1,1)	S(0,17)	|M(1,1) – S(1,1)|	|M(1,0) – S(1,1)|	|M(0,15) – S(0,17)|

FIGURE 7.32
A schedule of pixel fetches for a full search [Yan89].



bus.We may want to verify the operation of the basic PCI interface before we finish implementing the video accelerator logic The host PC will probably deal with the accelerator as an I/O device. The accelerator board will have its own driver that is responsible for talking to the board. Since most of the data transfers are performed directly by the board using DMA,the driver can be relatively simple.

System Testing:

Testing video algorithms requires a large amount of data. Luckily,the data represents images and video, which are plentiful. Because we are designing only a motion estimation accelerator and not a complete video compressor, it is probably easiest to use images, not video, for test data.You can use standard video tools to extract a few frames from a digitized video and store them in JPEG format. Open source for JPEG encoders and decoders is available. These programs can be modified to read
JPEG images and put out pixels in the format required by your accelerator. With a little more cleverness, the resulting motion vector can be written back onto the image for a visual confirmation of the result. If you want to be adventurous and try motion estimation on video,open source MPEG encoders and decoders are also available.