Sri Chandrasekharendra Saraswathi Viswa Maha Vidyalaya
Department of Electronics and Communication Engineering

STUDY MATERIAL

for

III rd Year

VI th Semester

Subject Name: EMBEDDED SYSTEMS

Prepared by

Dr.P.Venkatesan,
Associate Professor, ECE,
SCSVMV University

PRE-REQUISITE:

- Basic knowledge of Microprocessors, Microcontrollers & Digital System Design

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 and overview of real time Operating system.
- Learn the system design techniques and networks for embedded systems to industrial applications.

SYLLABUS

UNIT – I

INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS
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 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 Introduction – Multiple tasks and multiple processes – Multirate systems- Preemptive real-time operating systems-Priority based scheduling- Interprocess 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 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 Data compressor - Alarm Clock - Audio player - Software modem-Digital still camera - Telephone answering machine-Engine control unit – Video accelerator.

OUTCOMES:

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

1. Describe the architecture and programming of ARM processor.
2. Outline the concepts of embedded systems.
3. Use the system design techniques to develop software for embedded systems.
4. Differentiate between the general purpose and real time operating system.
5. Model real-time consumer/industrial applications using embedded-system concepts.

TEXT BOOKS:

1. Marilyn Wolf, “Computers as Components - Principles of Embedded Computing System Design”, Third Edition “Morgan Kaufmann Publisher, 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, 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, McGraw 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”, McGraw Hill, 2004.

EMBEDDED SYSTEMS

UNIT I

EMBEDDED COMPUTING

1.1 CHALLENGES OF EMBEDDED SYSTEMS:

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

How much hardware do we need?

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

How do we meet deadlines?

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

How do we minimize power consumption?

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

How do we design for upgradability?

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

Does it really work?

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

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

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

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

1.2 EMBEDDED SYSTEM DESIGN PROCESS:

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. Third, a design methodology makes it much easier for members of a design team to communicate. By defining the overall process, team members can more easily understand what they are supposed to do, what they should receive from other team members at certain times, and what they are to hand off when they complete their assigned steps. Since most embedded systems are designed by teams, coordination is perhaps the most important role of a well-defined design methodology.

Figure 1.1 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,

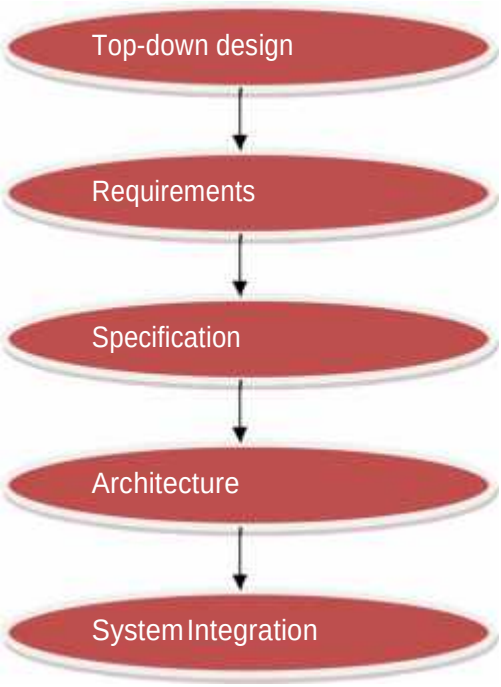


FIGURE 1.1
Major levels of abstraction in the design process.

specification, we create a more detailed description of what we want. But the specification states only how the system behaves, not how it is built. The details of the system's internals begin to take shape when we develop the architecture, which gives the system structure in terms of large components. Once we know the components we need, we can design those components, including both software modules and any specialized hardware we need. Based on those components, we can finally build a complete system. In this section we will consider design from the ***top–down***—we will begin with the most abstract description of the system and conclude with concrete details. The alternative is a **bottom–up** view in which we start with components to build a system. Bottom–up design steps are shown in the figure as dashed-line arrows. We need bottom–up design because we do not have perfect insight into how later stages of the design process will turn out. Decisions at one stage of design are based upon estimates of what will happen later:
How fast can we make a particular function run?
How much memory will we need?
How much system bus capacity do we need?
If our estimates are inadequate, we may have to backtrack and amend our original decisions to take the new facts into account. In general, the less experience we have with the design of similar systems , the more we will have to rely on bottom-up design information to help us refine the

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

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

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

- We must *analyze* the design at each step to determine how we can meet the specifications.
- We must then *refine* the design to add detail.
- And we must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

1.2.1 Requirements

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

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 ***nonfunctional***. We must of course capture the basic functions of the embedded system, but functional description is often not sufficient. Typical nonfunctional 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.
- C*ost:* 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 mock-up may use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation. But it should give the customer a good idea of how the system will be used and how the user can react to it. Physical, nonfunctional models of devices can also give customers a better idea of characteristics such as size and weight.

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

1.2.2 Specification

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

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

The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer. It should also be unambiguous enough that designers know what they need to build. Designers can run into several different types of problems caused by unclear specifications. If the behavior of some feature in a particular situation is unclear from the specification, the designer may implement the wrong functionality. If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

A specification of the GPS system would include several components:

- Data received from the GPS satellite constellation.

- Map data.
- User interface.
- Operations that must be performed to satisfy customer requests.
- Background actions required to keep the system running, such as operating the GPS receiver.

1.2.3 Architecture Design

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

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

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

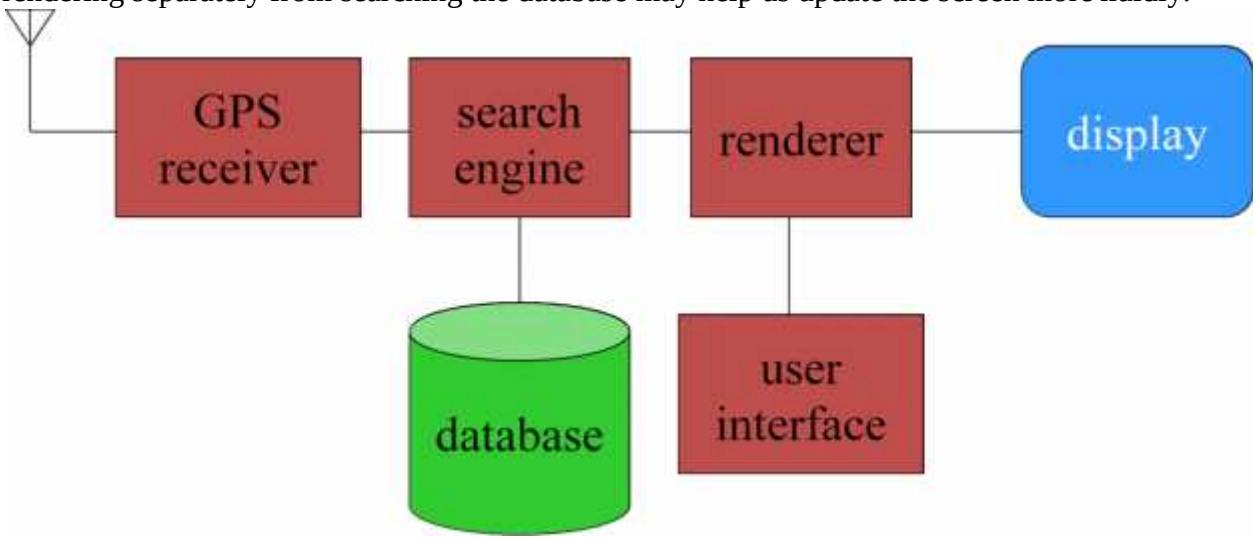
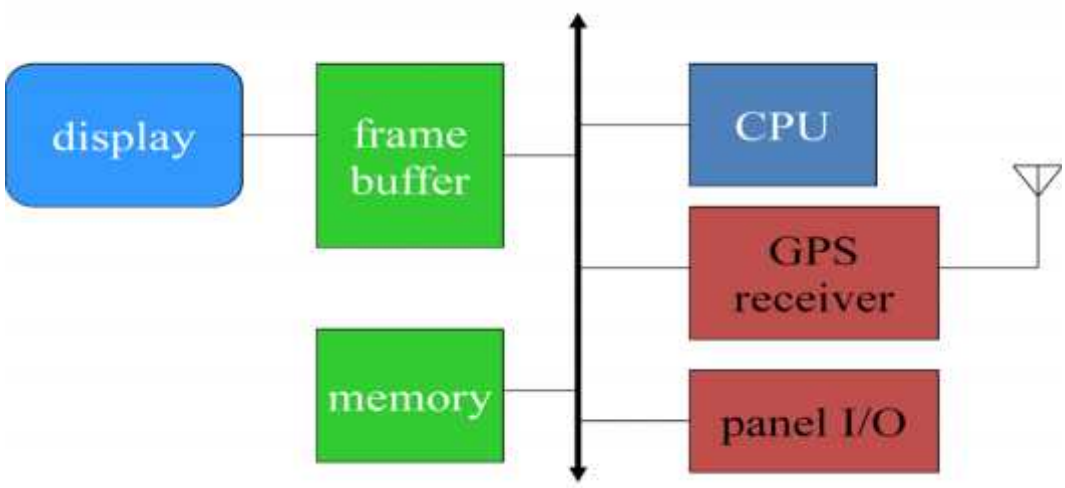
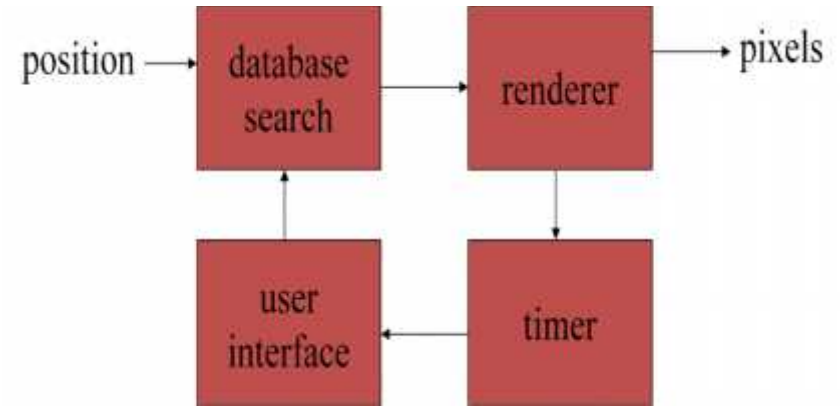


FIGURE 1.3 BLOCK DIAGRAM FOR THE MOVING MAP



HARDWARE



SOFTWARE

FIG 1.4 HARDWARE AND SOFTWARE ARCHITECTURES FOR THE MOVING MAP

1.2.4 DESIGNING HARDWARE AND SOFTWARE COMPONENTS:

- Must spend time architecting the system before you start coding.
- Some components are ready-made, some can be modified from existing designs, others must be designed from scratch.

1.2.5 SYSTEM INTEGRATION

- Put together the components.
  - Many bugs appear only at this stage.
- Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible.

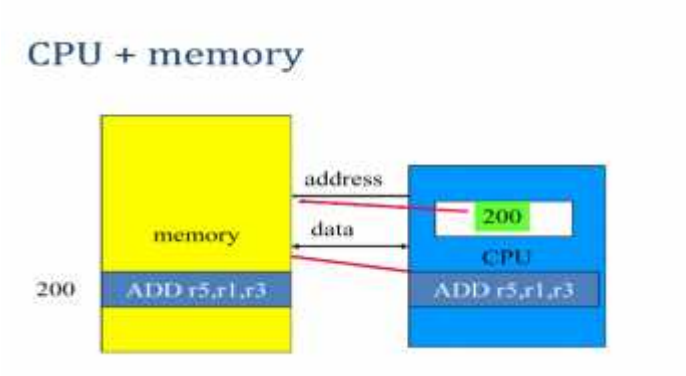
1.3 EMBEDDED PROCESSOR -8051 MICROCONTROLLER

1.4 ARM PROCESSOR

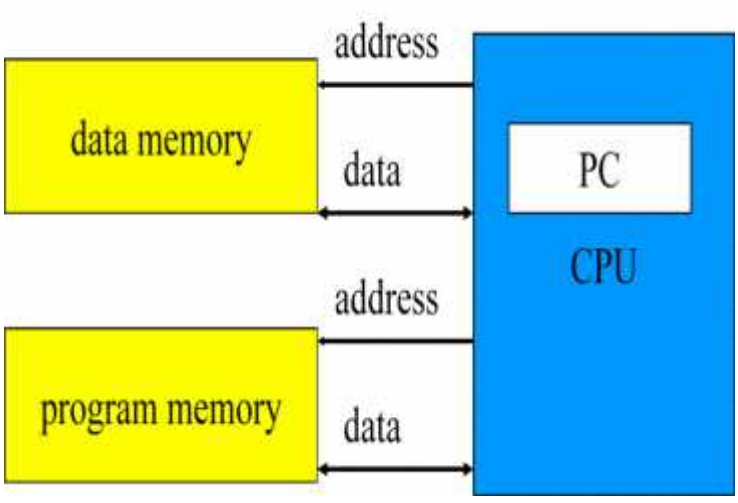
ARCHITECTURE:

VON NEUMANN ARCHITECTURE:

- Memory holds data, instructions.
- Central processing unit (CPU) fetches instructions from memory.
- Separate CPU and memory distinguishes programmable computer.
- CPU registers help out: program counter (PC), instruction register (IR), general-purpose registers, etc.



HARVARD ARCHITECTURE:



VON NEUMANN VS. HARVARD:

- Harvard can't use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth

INSTRUCTION SET CHARACTERISTICS:

- Fixed vs. variable length.
- Addressing modes.
- Number of operands.
- Types of operands.

Data Operations

Arithmetic and logical operations in C are performed in variables. Variables are implemented as memory locations. Therefore, to be able to write instructions to perform C expressions and assignments, we must consider both arithmetic and logical instructions as well as instructions for reading and writing memory. Figure 2.7 shows a sample fragment of C code with data declarations and several assignment statements. The variables *a*, *b*, *c*, *x*, *y*, and *z* all become data locations in memory. In most cases data are kept relatively separate from instructions in the program's memory image.

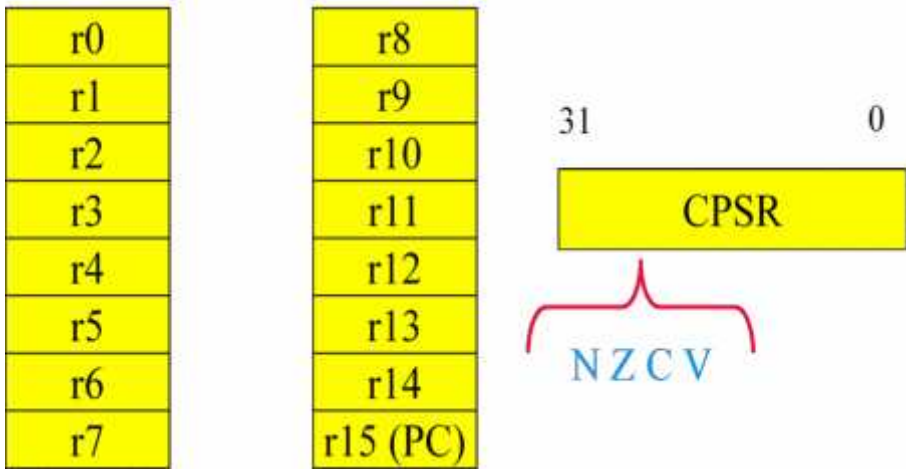
In the ARM processor, arithmetic and logical operations cannot be performed directly on memory locations. While some processors allow such operations to directly reference main memory, ARM is a ***load-store architecture***—data operands must first be loaded into the CPU and then stored back to main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

The other important basic register in the programming model is the ***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);
```

FIGURE 2.7
A C fragment with data operations.



- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
    - Address 4 starts at byte 4.
- Can be configured at power-up as either little- or bit-endian mode.

ARM STATUS BITS

- Every arithmetic, logical, or shifting operation sets CPSR bits:
    - N (negative), Z (zero), C (carry), V (overflow).
- Examples:
    - -1 + 1 = 0: NZCV = 0110.
    - 2^{31}-1+1 = -2^{31}: NZCV = 0101.

ARM DATA INSTRUCTIONS

- Basic format:
- ADD r0,r1,r2
    - Computes r1+r2, stores in r0.
- Immediate operand:
- ADD r0,r1,#2

- Computes r1+2, stores in r0.
- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
- MUL, MLA : multiply (and accumulate)
- AND, ORR, EOR
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C

DATA OPERATION VARIETIES:

- Logical shift:
  - fills with zeroes.
- Arithmetic shift:
  - fills with ones.
- RRX performs 33-bit rotate, including C bit from CPSR above sign bit.
- CMP : compare
- CMN : negated compare
- TST : bit-wise test
- TEQ : bit-wise negated test
- These instructions set only the NZCV bits of CPSR.
- MOV, MVN : move (negated)

  MOV r0, r1 ; sets r0 to r1

ARM LOAD/STORE INSTRUCTIONS:

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
  - register indirect : LDR r0,[r1]
  - with second register : LDR r0,[r1,-r2]
  - with constant : LDR r0,[r1,#4]

ARM ADR PSEUDO-OP

- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:

  ADR r1,FOO

PROGRAMMING:

- C:
  x = (a + b) - c;
- Assembler:

```
ADR r4,a        ; get address for a
LDR r0,[r4]     ; get value of a
ADR r4,b        ; get address for b, reusing r4
LDR r1,[r4]     ; get value of b
ADD r3,r0,r1    ; compute a+b
ADR r4,c        ; get address for c
LDR r2[r4]      ; get value of c
SUB r3,r3,r2    ; complete computation of x
ADR r4,x        ; get address for x
STR r3[r4]      ; store value of x
```

- C:
  y = a*(b+c);
- Assembler:

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MUL r2,r2,r0 ; compute final value for y
ADR r4,y ; get address for y
STR r2,[r4] ; store y
```

- C:
  z = (a << 2) | (b & 15);
- Assembler:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
ADR r4,z ; get address for z
STR r1,[r4] ; store value for z
```

ADDITIONAL ADDRESSING MODES:

- Base-plus-offset addressing:
  - LDR r0,[r1,#16]
  - Loads from location r1+16
- Auto-indexing increments base register:
  - LDR r0,[r1,#16]!
- Post-indexing fetches, then does offset:
  - LDR r0,[r1],#16
- Loads r0 from r1, then adds 16 to r1.

ARM FLOW OF CONTROL

- All operations can be performed conditionally, testing CPSR:
  - EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:
  B #100
    - Can be performed conditionally.

EXAMPLE: IF STATEMENT

- C:
  if (a > b) { x = 5; y = c + d; } else x = c - d;
- Assembler:

```
; compute and test condition
        ADR r4,a ; get address for a
        LDR r0,[r4] ; get value of a
        ADR r4,b ; get address for b
        LDR r1,[r4] ; get value for b
        CMP r0,r1 ; compare a < b
        BGE fblock ; if a >= b, branch to false block
; true block
        MOV r0,#5 ; generate value for x
        ADR r4,x ; get address for x
        STR r0,[r4] ; store x
        ADR r4,c ; get address for c
        LDR r0,[r4] ; get value of c
        ADR r4,d ; get address for d
        LDR r1,[r4] ; get value of d
        ADD r0,r0,r1 ; compute y
        ADR r4,y ; get address for y
        STR r0,[r4] ; store y
        B after ; branch around false block
; false block
fblock ADR r4,c ; get address for c
        LDR r0,[r4] ; get value of c
```

```
        ADR r4,d ; get address for d
        LDR r1,[r4] ; get value for d
        SUB r0,r0,r1 ; compute a-b
        ADR r4,x ; get address for x
        STR r0,[r4] ; store value of x
after ...
```

```
; true block
        MOVLT r0,#5 ; generate value for x
        ADRLT r4,x ; get address for x
        STRLT r0,[r4] ; store x
        ADRLT r4,c ; get address for c
        LDRLT r0,[r4] ; get value of c
        ADRLT r4,d ; get address for d
        LDRLT r1,[r4] ; get value of d
        ADDLT r0,r0,r1 ; compute y
        ADRLT r4,y ; get address for y
        STRLT r0,[r4] ; store y
```

EXAMPLE: SWITCH STATEMENT

- C:
  switch (test) { case 0: … break; case 1: … }
- Assembler:

```
        ADR r2,test ; get address for test
        LDR r0,[r2] ; load value for test
        ADR r1,switchtab ; load address for switch table
        LDR r1,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
        DCD case1
...
```

MEMORY AND INPUT / OUTPUT MANAGEMENT

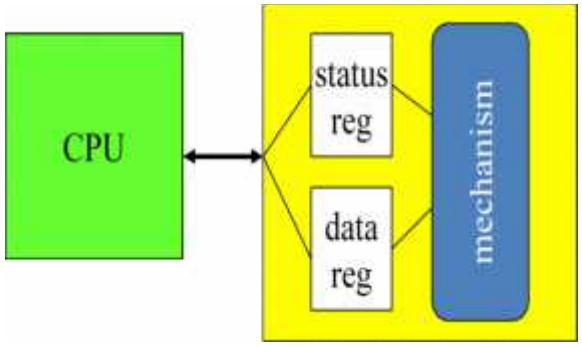
PROGRAMMING INPUT AND OUTPUT:

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 3.1 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. Application Example 3.1 describes a classic I/O device.



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. Example 3.1 illustrates memory-mapped I/O on the ARM.

Example 3.1

Memory-mapped I/O on ARM

We can use the EQU pseudo-op to define a symbolic name for the memory location of our I/O device:

```
DEV1 EQU 0x1000
```

Given that name, we can use the following standard code to read and write the device register:

```
LDR r1,#DEV1 ; set up device address
LDR r0,[r1] ; read DEV1
LDR r0,#8 ; set up value to write
STR r0,[r1] ; write 8 to device
```

The peek function can be written in C as:

```
int peek(char *location) {
return *location; /* de-reference location pointer */
}
```

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

```
#define DEV1 0x1000
...
dev_status = peek(DEV1); /* read device register */
```

The poke function can be implemented as:

```
void poke(char *location, char newval) {
(*location) = newval; /* write to location */
}
```

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

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

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

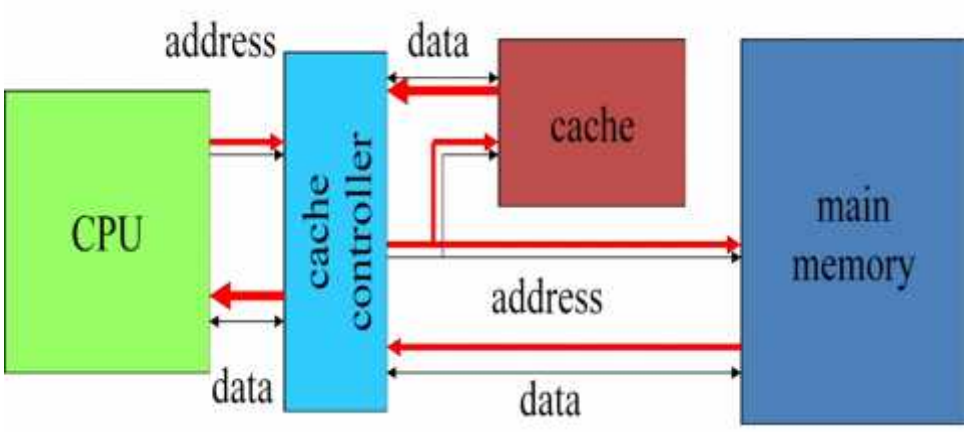
Busy-Wait I/O

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 befor 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***.

MEMORY SYSTEM MECHANISMS:

- **Caches.**
- **Memory management.**

CACHES AND CPUS



CACHE OPERATION

- Many main memory locations are mapped onto one cache entry.
- May have caches for:
  - instructions;
  - data;
  - data + instructions (unified).
- Memory access time is no longer deterministic.

- Cache hit: required location is in cache.
- Cache miss: required location is not in cache.
- Working set: set of locations used by program in a time interval.

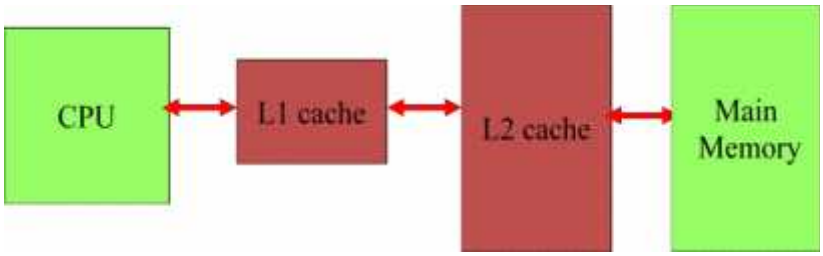
TYPES OF MISSES

- Compulsory (cold): location has never been accessed.
- Capacity: working set is too large.
- Conflict: multiple locations in working set map to same cache entry.

MEMORY SYSTEM PERFORMANCE

- h = cache hit rate.
- t_{cache} = cache access time, t_{main} = main memory access time.
- Average memory access time:
    - $t_{av} = ht_{cache} + (1-h)t_{main}$

MULTIPLE LEVELS OF CACHE



MULTI-LEVEL CACHE ACCESS TIME

- h_1 = cache hit rate.
- h_2 = hit rate on L2.
- Average memory access time:
    - $t_{av} = h_1t_{L1} + (h_2-h_1)t_{L2} + (1- h_2-h_1)t_{main}$

- Replacement policy: strategy for choosing which cache entry to throw out to make room for a new memory location.
- Two popular strategies:
    - Random.
    - Least-recently used (LRU).

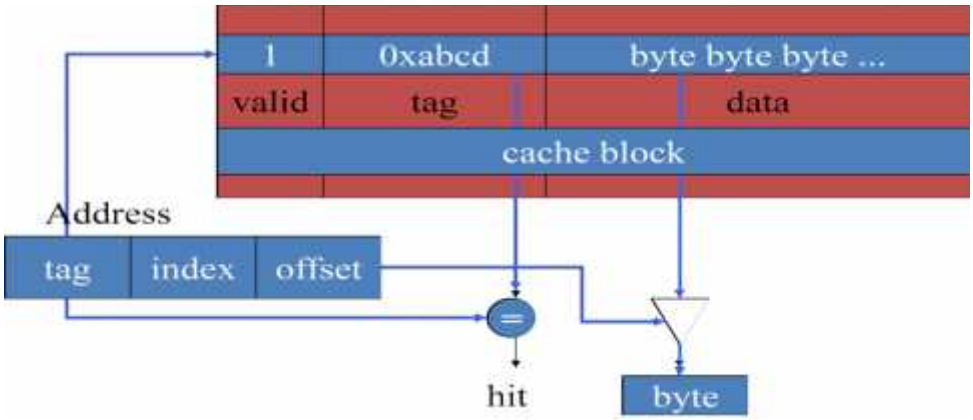
CACHE ORGANIZATIONS

- Fully-associative: any memory location can be stored anywhere in the cache (almost never implemented).
- Direct-mapped: each memory location maps onto exactly one cache entry.
- N-way set-associative: each memory location can go into one of n sets.

CACHE PERFORMANCE BENEFITS

- Keep frequently-accessed locations in fast cache.
- Cache retrieves more than one word at a time.
    - Sequential accesses are faster after first access.

DIRECT-MAPPED CACHE



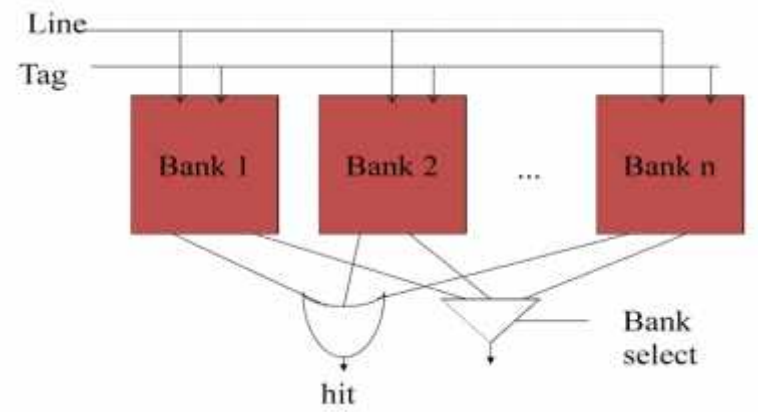
WRITE OPERATIONS

- Write-through: immediately copy write to main memory.
- Write-back: write to main memory only when location is removed from cache.

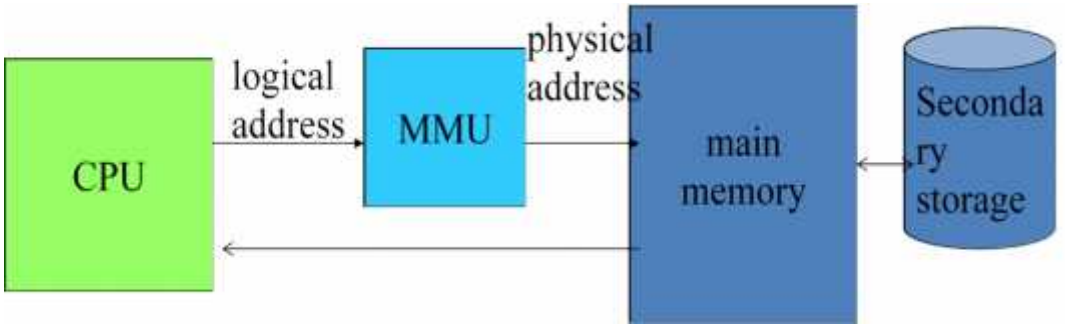
DIRECT-MAPPED CACHE LOCATIONS

- Many locations map onto the same cache block.
- Conflict misses are easy to generate:
    - Array a[] uses locations 0, 1, 2, …
    - Array b[] uses locations 1024, 1025, 1026, …
    - Operation a[i] + b[i] generates conflict misses.

SET-ASSOCIATIVE CACHE

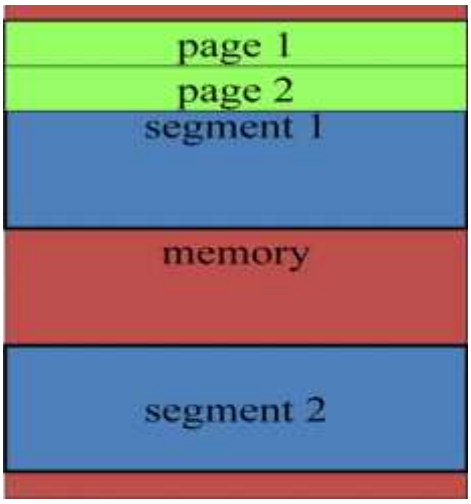


MEMORY MANAGEMENT UNITS

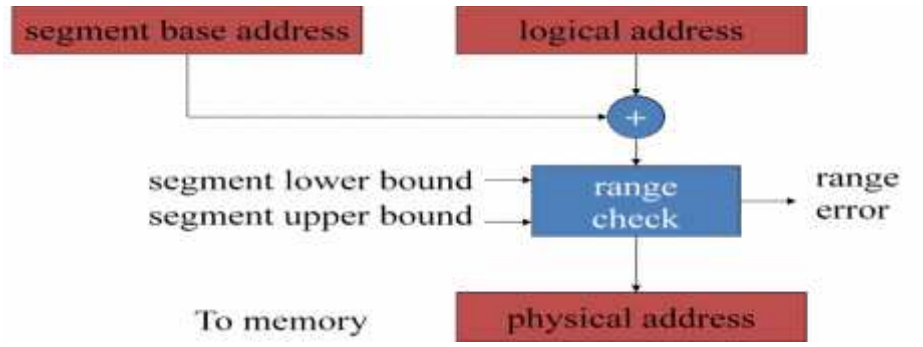


- Allows programs to move in physical memory during execution.
- Allows virtual memory:
  - memory images kept in secondary storage;
  - images returned to main memory on demand during execution.
- Page fault: request for location not resident in main memory.
- Requires some sort of register/table to allow arbitrary mappings of logical to physical addresses.
- Two basic schemes:
  - segmented;
  - paged.
- Segmentation and paging can be combined (x86).

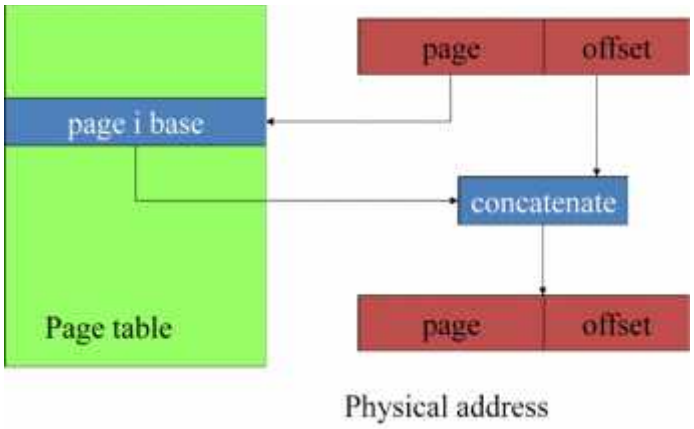
SEGMENTS AND PAGES



SEGMENT ADDRESS TRANSLATION

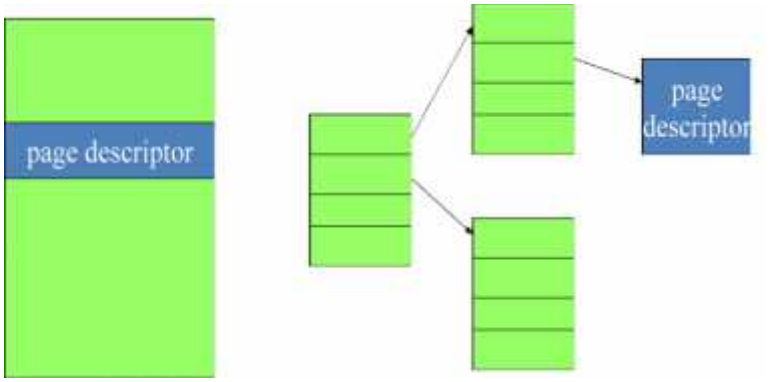


PAGE ADDRESS TRANSLATION



- Logical address is divided into two sections including a page number and an offset
- The page no is used as an index into a page table, which stores the physical address for the start of each page
- The page table is kept in is normally kept in main memory, the address translation requires memory access.

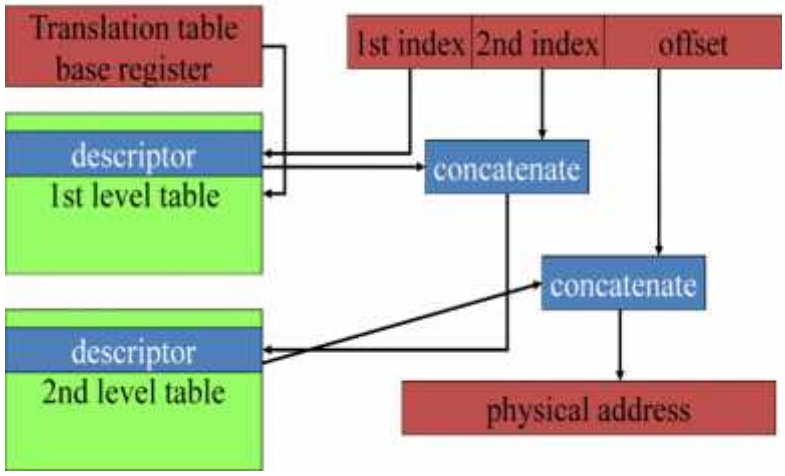
PAGE TABLE ORGANIZATIONS



CACHING ADDRESS TRANSLATIONS

- Large translation tables require main memory access.
- TLB: cache for address translation.
    - Typically small.
- TLB – Translation lookaside buffer

ARM ADDRESS TRANSLATION



MEMORY AND I/O DEVICES AND INTERFACING

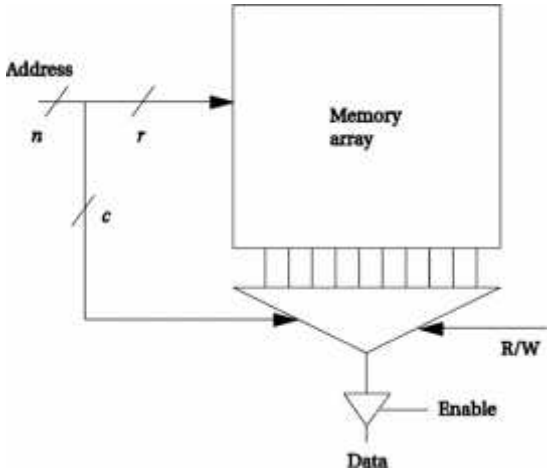
- Memory devices.
- I/O devices:
    - serial links

- timers and counters
- keyboards
- displays
- analog I/O

MEMORY DEVICES.

MEMORY COMPONENTS

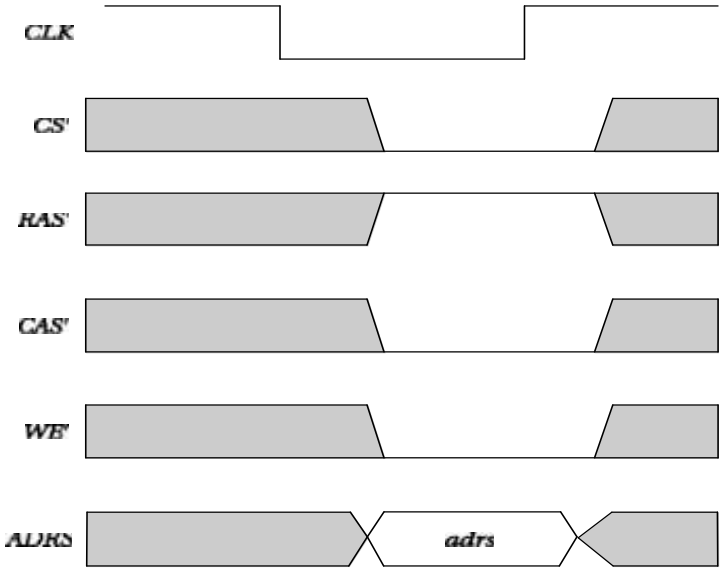
- Several different types of memory:
  - DRAM.
  - SRAM.
  - Flash.
- Each type of memory comes in varying:
  - Capacities.
  - Widths.



RANDOM-ACCESS MEMORY

- Dynamic RAM is dense, requires refresh.
  - Synchronous DRAM is dominant type.
  - SDRAM uses clock to improve performance, pipeline memory accesses.
- Static RAM is faster, less dense, consumes more power.

SDRAM OPERATION



READ-ONLY MEMORY

- ROM may be programmed at factory.
- Flash is dominant form of field-programmable ROM.
    - Electrically erasable, must be block erased.
    - Random access, but write/erase is much slower than read.
    - NOR flash is more flexible.
- NAND flash is more dense.

FLASH MEMORY

- Non-volatile memory.
    - Flash can be programmed in-circuit.
- Random access for read.
- To write:
    - Erase a block to 1.
    - Write bits to 0.
    - Write is much slower than read.
    - ms write, 70 ns read.
    - Blocks are large (approx. 1 Mb).
    - Writing causes wear that eventually destroys the device.
    - Modern lifetime approx. 1 million writes.

TYPES OF FLASH

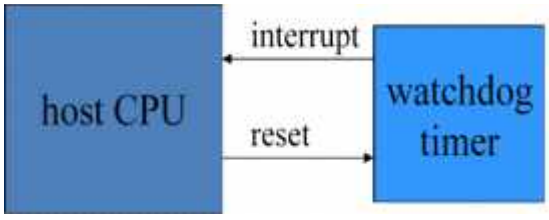
- NOR:
  - Word-accessible read.
  - Erase by blocks.
- NAND:
  - Read by pages (512-4K bytes).
  - Erase by blocks.
- NAND is cheaper, has faster erase, sequential access times.

TIMERS AND COUNTERS

- Very similar:
  - a timer is incremented by a periodic signal;
  - a counter is incremented by an asynchronous, occasional signal.
- Rollover causes interrupt.

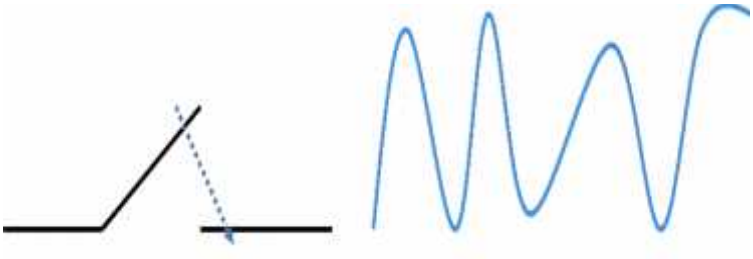
WATCH DOG TIMER

- Watchdog timer is periodically reset by system timer.
- If watchdog is not reset, it generates an interrupt to reset the host.



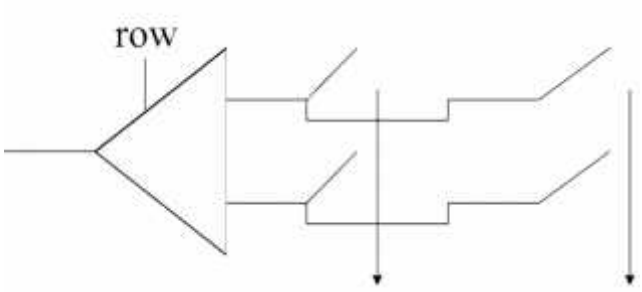
SWITCH DEBOUNCING

- A switch must be debounced to multiple contacts caused by eliminate mechanical bouncing:

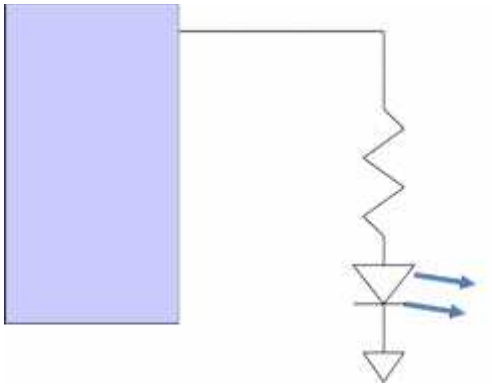


ENCODED KEYBOARD

- An array of switches is read by an encoder.
- N-key rollover remembers multiple key depressions.



LED



- Must use resistor to limit current:

7-SEGMENT LCD DISPLAY

- May use parallel or multiplexed input.

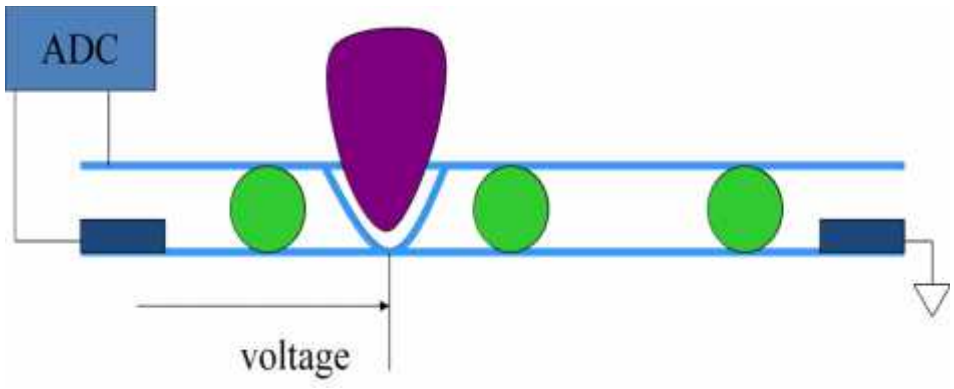
TYPES OF HIGH-RESOLUTION DISPLAY

- Liquid crystal display (LCD) is dominant form.
- Plasma, OLED, etc.
- Frame buffer holds current display contents.
- Written by processor.
- Read by video.

TOUCHSCREEN

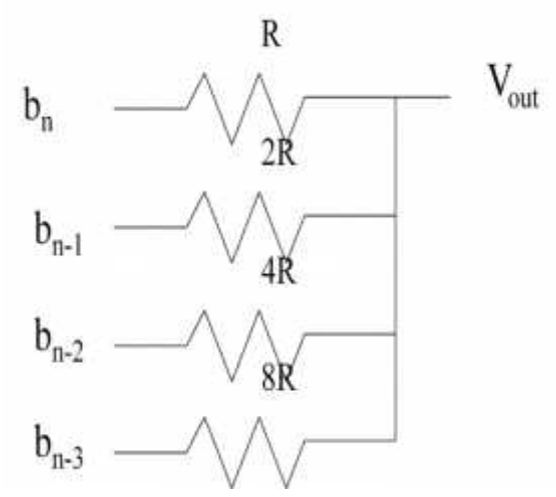
  - Includes input and output device.
- Input device is a two-dimensional voltmeter:

TOUCHSCREEN POSITION SENSING

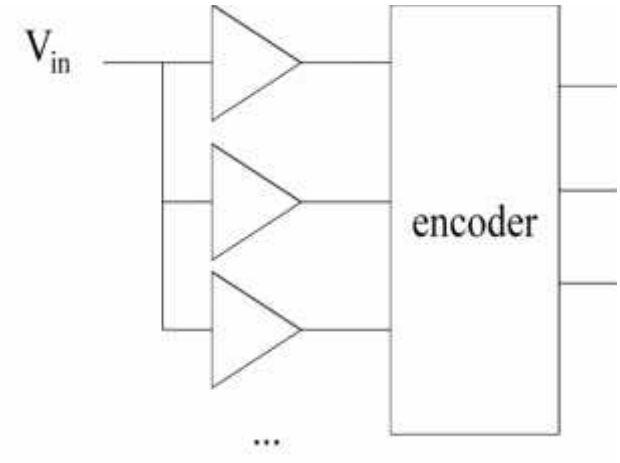


DIGITAL-TO-ANALOG CONVERSION

- Use resistor tree:

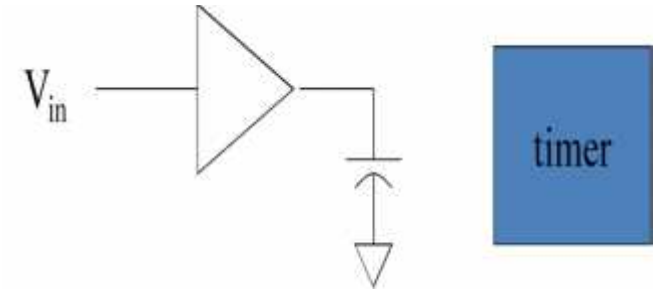


FLASH A/D CONVERSION



DUAL-SLOPE CONVERSION

- Use counter to time required to charge/discharge capacitor.
- Charging, then discharging eliminates non-linearities.



INTERRUPT HANDLING:

1. **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.

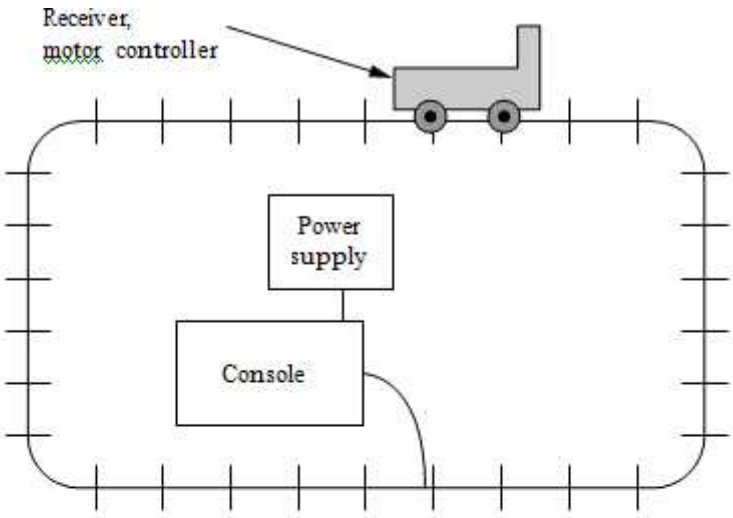
Requirements

- 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

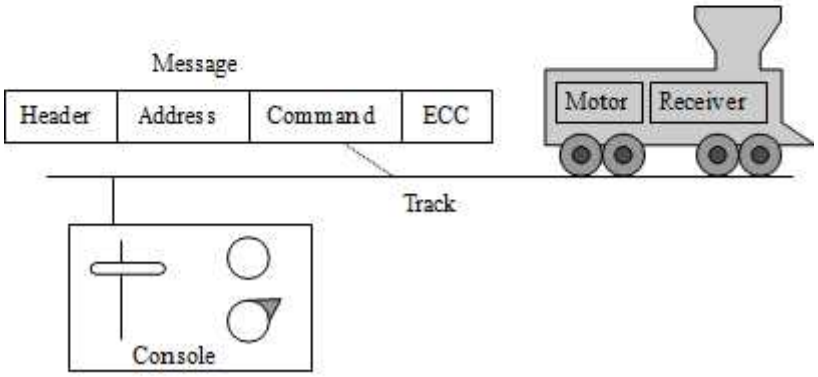
can put the requirements into chart format:

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	Can update train speed at least 10 times per second
Physical	$50 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.

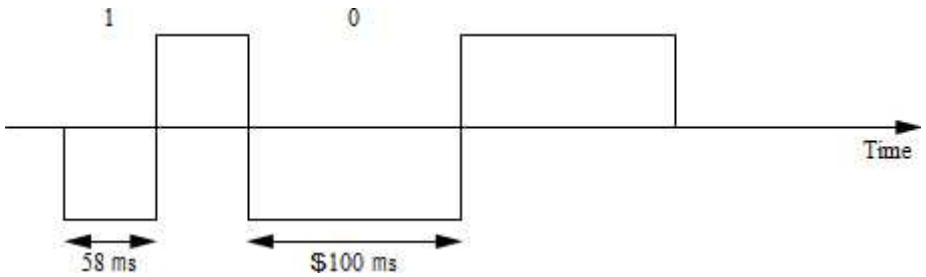
The data signal swings between two voltages around the power supply voltage. As shown in Figure , bits are encoded in the time between transitions, not by voltage levels. A 0 is at least 100 ms while a 1 is nominally 58ms.

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.

IMPORTANT QUESTIONS

PART-A(2 MARKS)

1. **State the importance of data register and status register.**
2. **Mention the two ways used for performing input and output operations.**
3. **Define polling.**
4. **Define an interrupt.**
5. **Mention the signals used for by i/o devices for interrupting.**
6. **Define foreground program.**
7. **Mention the ways used for generalizing the interrupts to handle multiple devices.**
8. **Define non maskable interrupts.**
9. **Define exception.**
10. **Define trap.**
11. **Define cache memory.**
12. **Define cache hit.**
13. **Define cache miss.**
14. **Mention the types of cache misses.**
15. **Mention the different strategies used for writing in a cache.**
16. **Define page fault.**
17. **Define DMA.**
18. **Mention the registers present in the DMA controller.**
19. **What is a watch dog timer?**
20. **Define aspect ratio.**
21. **What is an embedded computer system?**
22. **Why do we use microprocessors to design a digital system?**
23. **Mention the challenges in embedded computing system design.**
24. **Mention the reasons that makes embedded computing machines design difficult.**
25. **State the importance of design methodology.**
26. **Mention the major steps in embedded system design process.**
27. **Mention the major goals of embedded system design.**
28. **Mention the non functional requirements.**
29. **Mention the components of GPS system specification.**
30. **Mention the different types of relationships.**
31. **What is called a von Neumann machine?**
32. **What is called a Harvard machine?**
33. **Mention the characteristics of instructions.**
34. **State the importance of current program status register (CPSR).**
35. **Mention the addressing modes of C55x DSP.**
36. **Define procedure linkage.**
37. **Define CISC.**
38. **Define RISC.**
39. **Mention the features of assembly language.**
40. **Differentiate big and little endian byte ordering modes.**

PART-B(16 MARKS)

1. **Explain the concept of interrupts in detail.**
2. **Explain the working of cache memory in detail.**
3. **Explain memory mapping and address translation in detail.**
4. **Explain the working of CPU bus in detail.**
5. **Explain direct memory access in detail.**
6. **Explain the various I/O devices in detail.**
7. **Explain memory devices in detail.**
8. **Explain the various display devices and the methods of interfacing in detail.**
9. **Explain about exceptions and trap in detail.**
10. **Explain about interrupt priority and vector in detail**
11. **Explain in detail about the challenges in embedded computing system design.**
12. **Explain in detail about the embedded system design process.**
13. **Explain in detail about ARM processor.**
14. **Explain in detail about TI C55x DSP.**
15. **Explain in detail about the characteristics of embedded computing applications.**
16. **Explain Structural description in detail.**
17. **Explain Behavioral description in detail.**
18. **.Explain Conceptual specification in detail.**
19. **Explain in detail about 8051 microcontroller.**
20. **Explain in detail about data operations of ARM processor.**

EMBEDDED SYSTEMS
UNIT II

EMBEDDED COMPUTING PLATFORM DESIGN

CPU BUSES:

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.

Bus Organization and Protocol:

A bus is a common connection between components in a system.in figure 2.1 shows organization of a bus. The CPU, memory and I/O devices are all connected to the bus. The signals that make up the bus provide the necessary communication: Clock, Control, Address and Data.

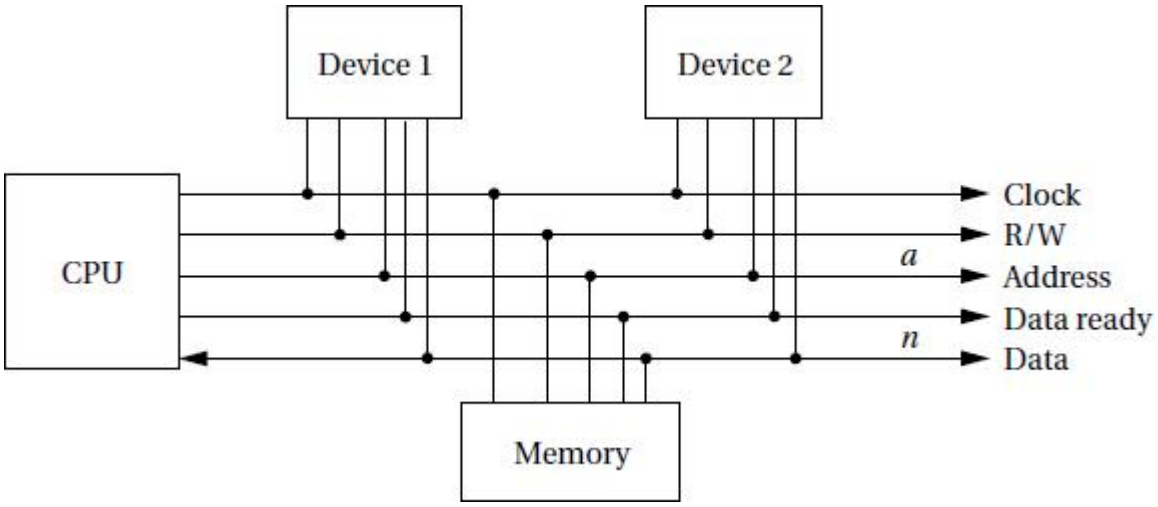


Fig 2.1 Organization of Bus.

Bus Master: In a typical bus system ,the CPU serves as the Bus master and initiates all transfer.

Four cycle handshake: The basic building block of most bus protocols is the ***four-cycle handshake***, illustrated in Figure 2.2. 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.

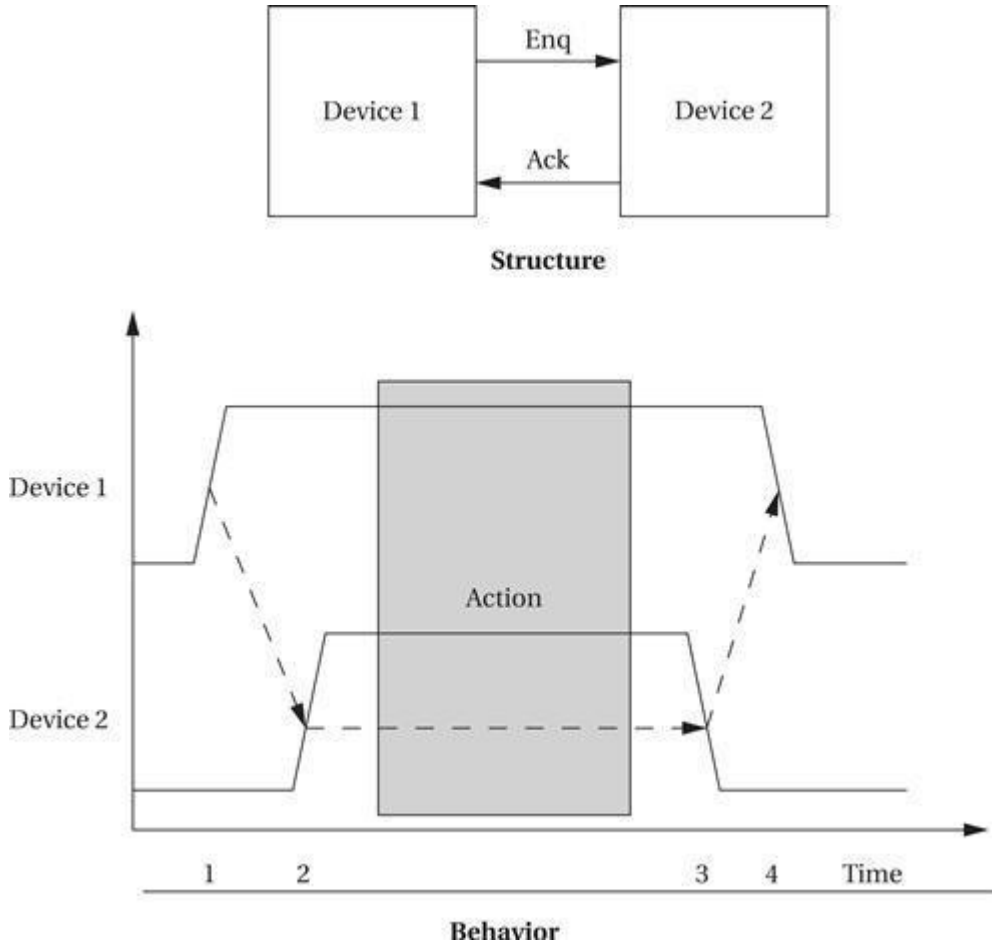


Fig 2.2 The four-cycle handshake.

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 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,

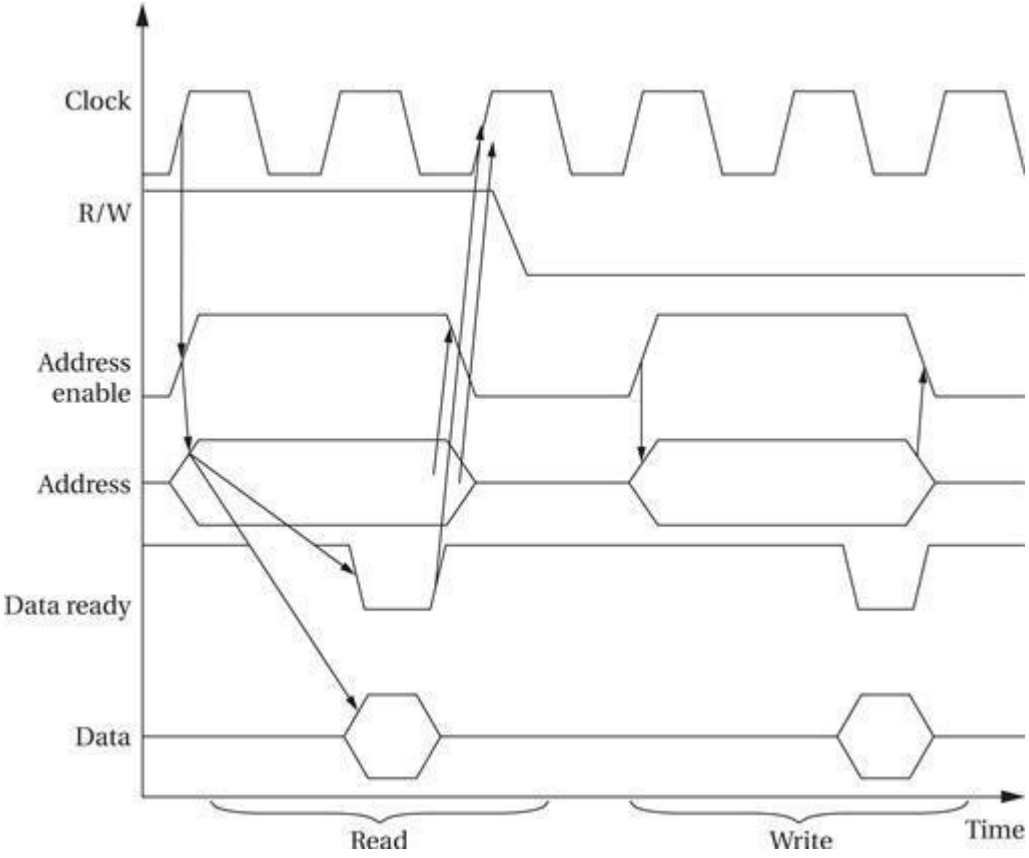
■ *Address* is an *a*-bit bundle of signals that transmits the address for an access,

■ *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.

Figure 2.3 shows a timing diagram for the example bus. The diagram shows a read and a write. Timing constraints are shown only for the read operation, but similar constraints apply to the write operation. The bus is normally in the read mode since that does not change the state of any of the devices or memories.

The CPU can then ignore the bus data lines until it wants to use the results of a read. Notice also that the direction of data transfer on bidirectional lines is not specified in the timing diagram.



2.3 Timing diagram for the example 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.

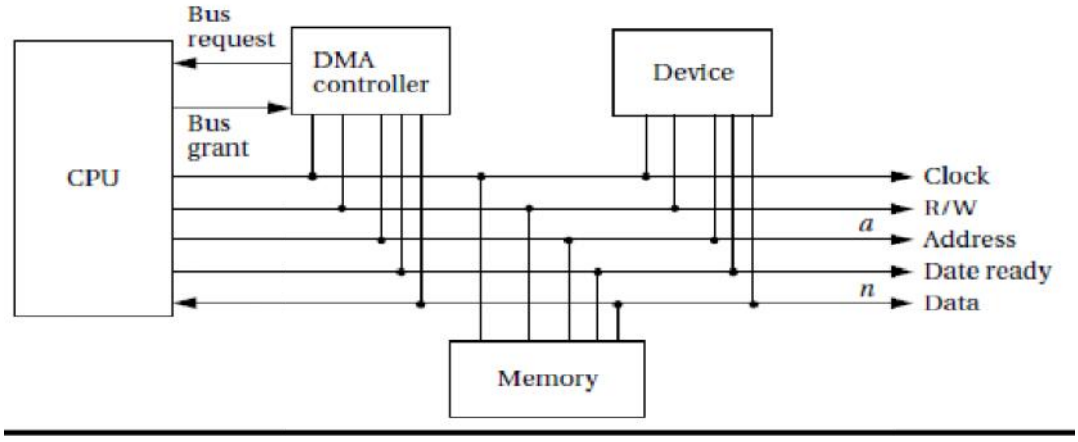


Fig 2.4 A bus with a DMA controller.

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 2.4 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).

System Bus Configurations:

A microprocessor system often has more than one bus. As shown in Figure 2.5, high-speed devices may be connected to a high-performance bus, while lower-speed devices are connected to a different bus. A small block of logic known as a ***bridge*** allows the buses to connect to each other. There are several good reasons to use multiple buses and bridges:

- Higher-speed buses may provide wider data connections.
- A high-speed bus usually requires more expensive circuits and connectors. The cost of low-speed devices can be held down by using a lower-speed, lower-cost bus.
- The bridge may allow the buses to operate independently, thereby providing some parallelism in I/O operations.

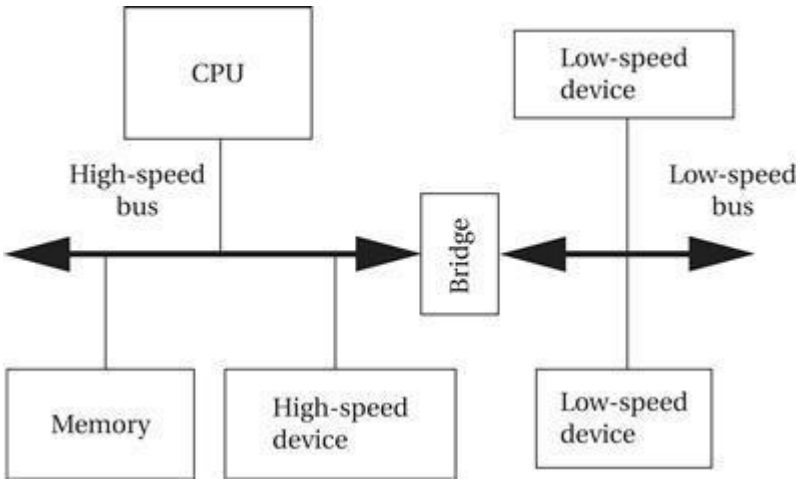


Figure 2.5 Multiple Bus system

AMBA Bus

Since the ARM CPU is manufactured by many different vendors, the bus provided off-chip can vary from chip to chip. ARM has created a separate bus specification for single-chip systems.

The AMBA bus [ARM99A] supports CPUs, memories, and peripherals integrated in a system-on-silicon. As shown in Figure 2.6,

the AMBA specification includes two buses. The AMBA high-performance bus (AHB) is optimized for high-speed transfers and is directly connected to the CPU. It supports several high-performance features: pipelining, burst transfers, split transactions ,and multiple bus masters.

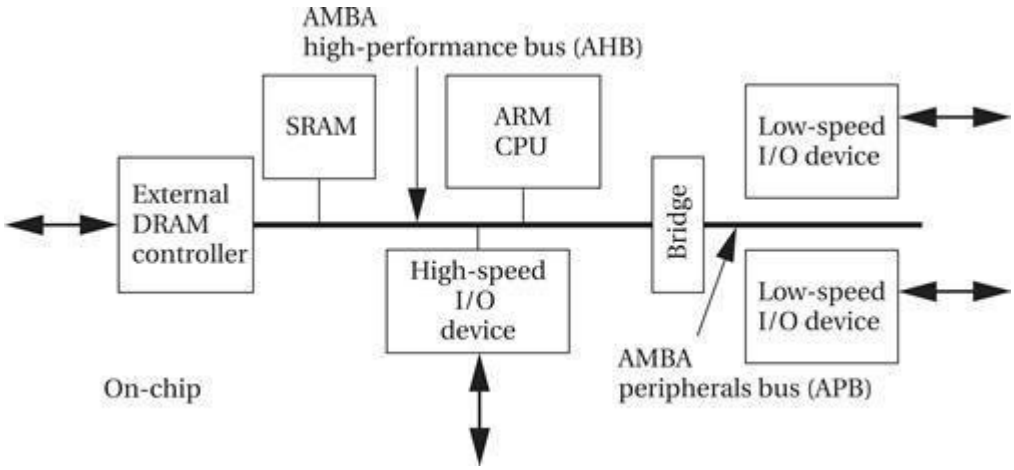


Figure 2.6, ARM-AMBA bus sytem

A bridge can be used to connect the AHB to an AMBA peripherals bus (APB). This bus is designed to be simple and easy to implement; it also consumes relatively little power. The AHB assumes that all peripherals act as slaves, simplifying the logic required in both the peripherals and the bus controller. It also does not perform pipelined operations, which simplifies the bus logic.

MEMORY DEVICES:

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 2^{26} different addresses.
- As a 32 M* 8-bit array, a single memory access obtains a 1-bit data item, with a maximum of 2^{23} 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 2.7. 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 $n = r + c$).

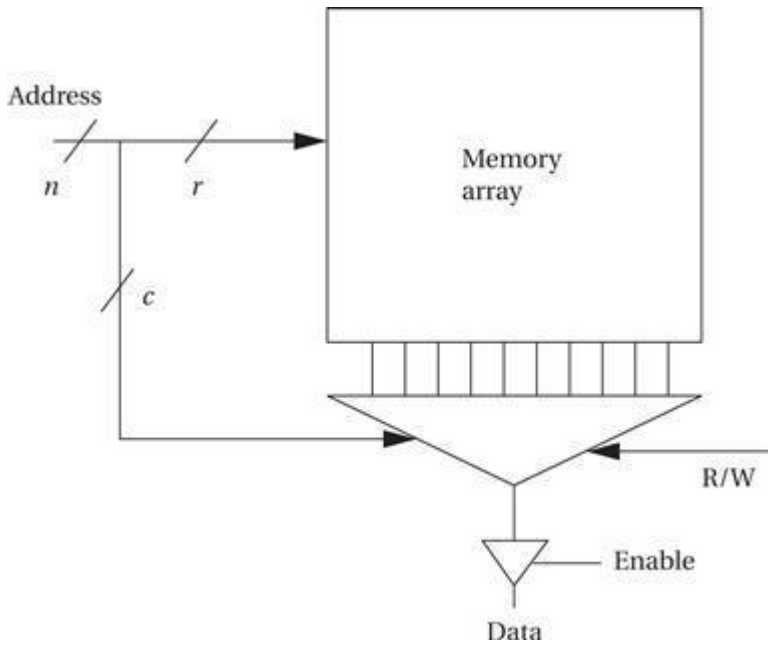


Fig 2.7 Internal organization of a memory device.

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 overtime.

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.

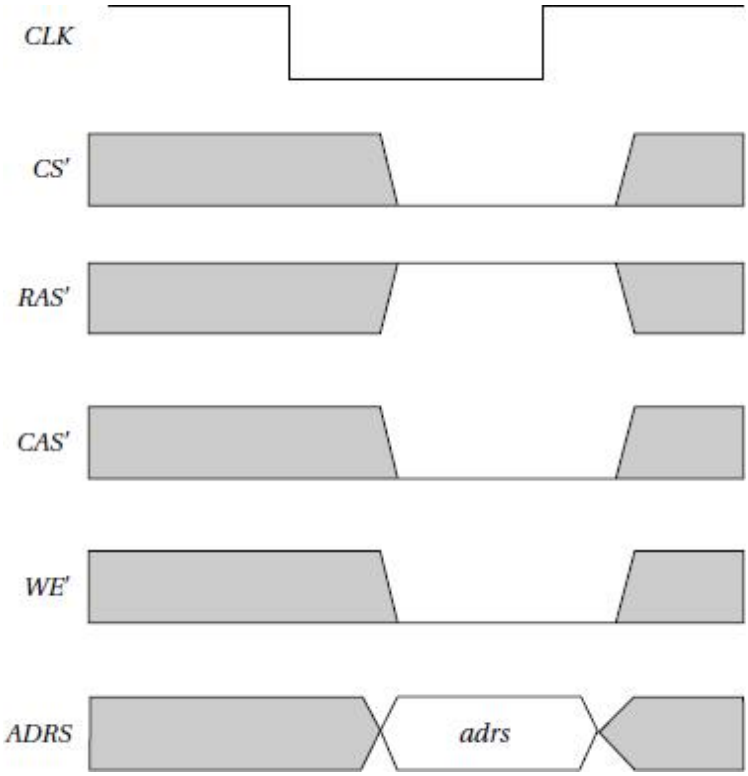


Fig 2.8 Timing diagram for a read on a synchronous DRAM.

As shown in Figure 2.8, transitions on the control signals are related to a clock. 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***.

DESIGN WITH COMPUTING PLATFORMS

Choosing Platform:

We may assemble hardware and software components from several sources to construct embedded system platform.

Hardware:

The hardware architecture of an embedded computing system includes several elements, some of which may be less obvious than others.

1. **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.

2. **Bus** :The choice of a bus is closely tied to that of a CPU, since the bus is an integral part of the microprocessor. Attention must be paid to the required data bandwidths to be sure that the bus can handle the traffic.

3. **Memory:** The most obvious characteristic of memory 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.

4. **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.

5. **Software:** Software components of the platform about both run time components and support components. Run time components includes the operating system, code libraries and so on.. Support components include the code development environment, debugging tools

Hardware Design

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

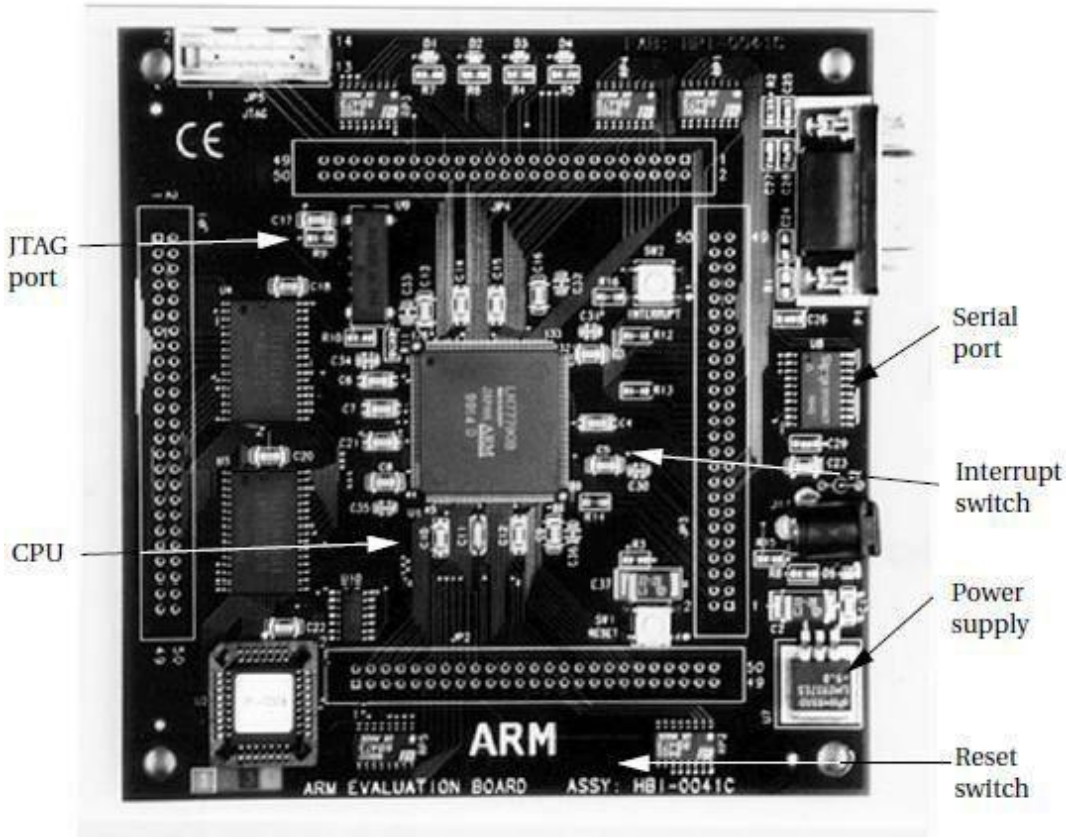


Fig 2.9 An ARM evaluation board.

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 2.9 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.

Intellectual property:

Intellectual property (IP) is something that we can own but not touch: software, netlist and so on. Just as we need to acquire hardware components to build our system ,we also need to acquire intellectual property to make the hardware useful.

Some wide range of IP that we use in embedded system design.

- Run time software libraries
- Software development environments
- Schematics ,netlist and other hardware design information.

Development Environments:

Although we may use an evaluation board , much of the software development for an embedded system is done on a PC or workstation known as a ***host*** as illustrated in Figure 2.10. 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.

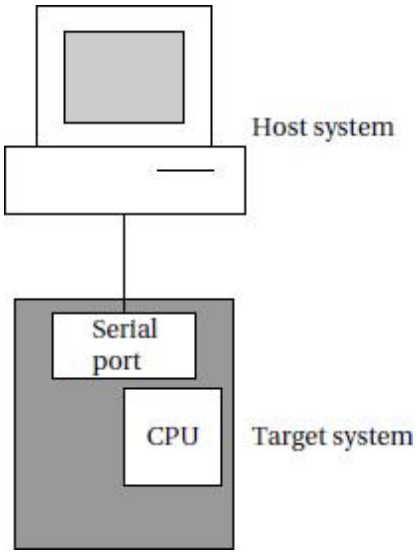


Fig 2.10 Connecting a host and a target system.

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,

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.

A ***cross-compiler*** is a compiler that runs on one type of machine but generates code for another. After compilation, the executable code is downloaded to the embedded system by a serial link or perhaps burned in a PROM and plugged in.

The ***testbench*** generates inputs to simulate the actions of the input devices; it may also take the output values and compare them against expected values, providing valuable early debugging help.

Debugging Techniques

A good deal of software debugging can be done by compiling and executing the code on a PC or workstation. the resourceful designer has several options available for debugging the system.

USB port: The **USB 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 USB 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.

Breakpoint: 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

LED: LEDs as debugging devices. As with serial ports, it is often a good idea to design a few to indicate the system state even if they will not normally be seen in use. LEDs can be used to show error conditions. a simple flashing LED can provide a great sense of accomplishment when it first starts to work.

In-Circuit emulator:

- When software tools are insufficient to debug the system, hardware aids can be deployed to give a clearer view of what is happening when the system is running. The ***microprocessor in-circuit emulator (ICE)*** is a

specialized hardware tool that can help debug software in a working embedded system.

- At the heart of an in-circuit emulator is a special version of the microprocessor that allows its internal registers to be read out when it is stopped.

- The main drawback to in-circuit emulation is that the machine is specific to a particular microprocessor, even down to the pin out. If you use several microprocessors, maintaining a fleet of in-circuit emulators to match can be very expensive.

Logic Analyzer: The logic analyzer as an array of in expensive oscilloscopes—the analyzer can sample many different signals simultaneously (tens to hundreds) but can display only 0, 1, or changing values for each.
All these logic analysis channels can be connected to the system to record the activity on many signals simultaneously. The logic analyzer records the values on the signals into an internal memory and then displays the results on a display once the memory is full or the run is aborted.

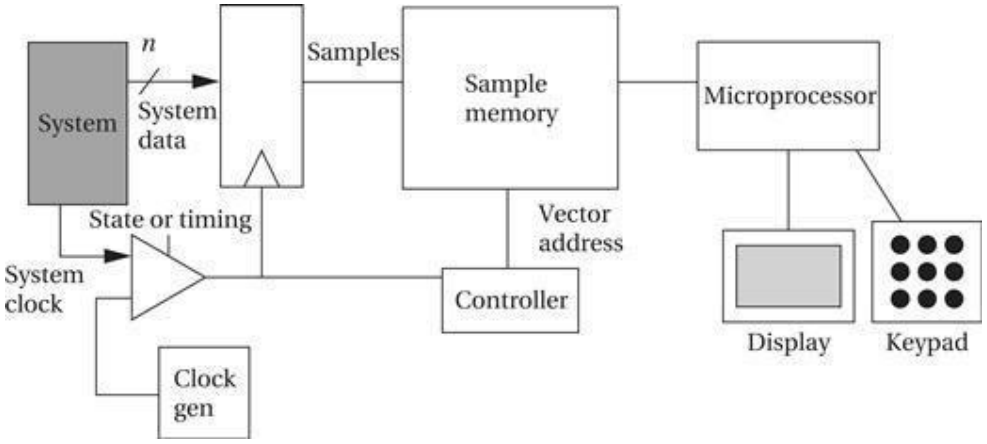
A typical logic analyzer can acquire data in either of two modes. ***state*** and ***timing modes***.

State and timing mode represent different ways of sampling the values.

Timing mode uses an internal clock that is fast enough to take several samples per clock period in a typical system. It requires more memory to store a given number of system clock cycles. As a result, On the other hand, it provides greater resolution in the signal for detecting glitches. Timing mode is typically used for glitch-oriented debugging.

State mode, on the other hand, uses the system's own clock to control sampling, so it samples each signal only once per clock cycle. state mode is used for sequentially oriented problems.

The internal architecture of a logic analyzer is shown in Figure 2.11.



2.11 Architecture of a logic analyzer.

The system's data signals are sampled at a **latch** within the logic analyzer; the latch is controlled by either the system clock or the internal logic analyzer sampling clock, depending on whether the analyzer is being used in state or timing mode.

Each sample is copied into a vector memory under the control of a state machine. The latch, timing circuitry, sample memory, and controller must be designed to run at high speed since several samples per system clock cycle may be required in timing mode.

After the sampling is complete, an embedded microprocessor takes over to control the display of the data captured in the sample memory.

Logic analyzers typically provide a number of formats for viewing data. One format is a timing diagram format. Many logic analyzers allow not only customized displays, such as giving names to signals, but also more advanced display options.

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.

Consumer Electronics Architecture:

In this section we consider consumer electronics devices as an example of complex embedded systems and the platforms that support them.

Consumer Electronics Use Cases a nd Requirements:

Although some predict the complete convergence of all 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. 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 functions in various combinations: 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.

Functional requirements:

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 DigitalTM, and so on for audio; JPEG for still images; MPEG-2, MPEG-4, H.264, and so on for video.

D**ata 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 a USB interface to a host computer. The communications link may also be more sophisticated, such as an Ethernet port or a cellular telephone link.

Nonfunctional requirements:

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 75 mW, 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

Use cases:

Let's consider some basic use cases of some basic operations. Figure 2.12 shows a use case for selecting and playing a multimedia object (an audio clip, a picture, etc.). Selecting 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.

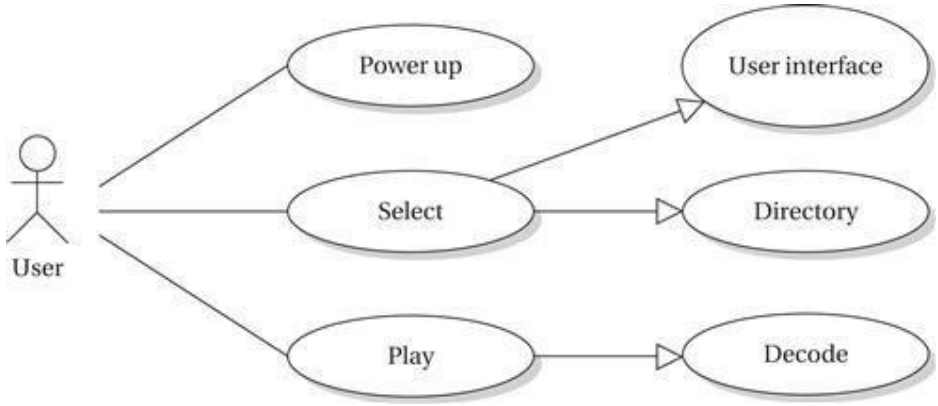


Figure 2.12 Use case for playing multimedia.

Figure 2.13 shows a use case for connecting to a client. 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.

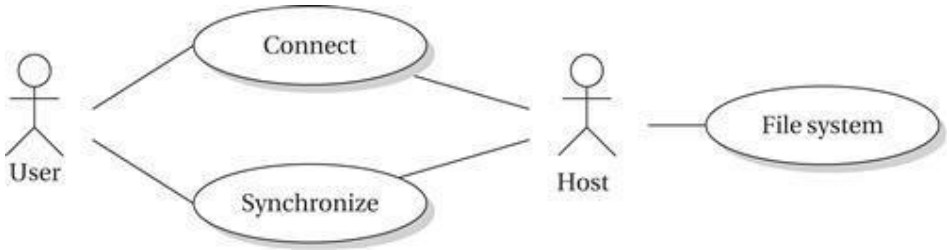


Figure 2.13 Use case of synchronizing with a host system.

Hardware architectures:

Figure 2.14 shows a functional block diagram of a typical device. The storage system provides bulk, permanent storage. The network interface may provide a simple USB connection or a full-blown Internet connection.

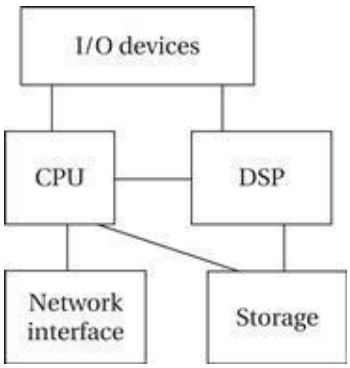


Figure 2.14 Hardware architecture of a generic consumer electronics device.

Multiprocessor architectures are common in many consumer multimedia devices. Figure 2.14 shows a two-processor architecture; if more computation is required, more DSPs and CPUs may be added.

The RISC CPU runs the operating 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 the complexity of the device, the operating system may not need to create tasks dynamically. If all tasks can be created using initialization code, the operating system can be made smaller and simpler.

FILE SYSTEMS

DOS file systems:

DOS file allocation table (**FAT**) file systems refer to the file system developed by Microsoft for early versions of the DOS operating system . FAT can be implemented on flash storage devices as well as magnetic disks; wear-leveling algorithms for flash memory can be implemented without disturbing the basic operation of the file system.

Flash memory:

Many consumer electronics devices use **flash memory** for mass storage. Flash memory is a type of semiconductor memory that, unlike DRAM or SRAM, provides permanent storage.

Values are stored in the flash memory cell as an electric charge using a specialized capacitor that can store the charge for years. The flash memory cell does not require an external power supply to maintain its value.

Furthermore, the memory can be written electrically and, unlike previous generations of electrically-erasable semiconductor memory, can be written using standard power supply voltages and so does not need to be disconnected during programming.

Platform-Level PerformanceAnalysis

Bus-based systems add another layer of complication to performance analysis.
Consider the simple system of Figure 2.15. 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.

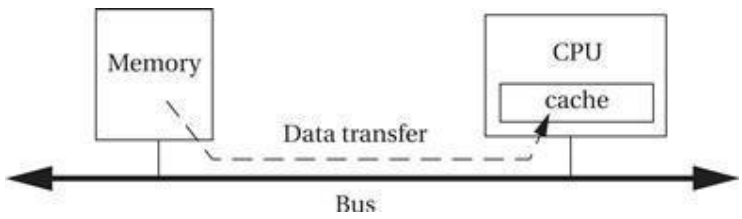


Figure 2.15 Platform-level data flows and performance.

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:

when we are transferring large blocks of data ,we consider Bus bandwidth.

Consider an image of 320 × 240 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 230,400 μs = 0.23 sec to transfer one frame. That is more than the 0.033 sec allotted to the data transfer.

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 0.029 sec, which is within our time budget for the transfer.

Bus bandwidth formulas:

Let's call the bus clock period *P* and the bus width *W*. We will put *W* in units of bytes but we could use other measures of width as well.

We want to write formulas for the time required to transfer *N* bytes of data. We will write our basic formulas in units of bus cycles *T*, then convert those bus cycle counts to real time *t* using the bus clock period *P*:

$$t=T\ P$$

Memory aspect ratio:

A single memory chip is not solely specified by the number of bits it can hold.

As shown in Figure 2.16, memories of the same size can have different **aspect ratios.**

For example, a 64-Mbit memory that is one bit wide will present 64 million addresses of one-bit data.

The same size memory in a 4-bit-wide format will have 16 distinct addresses and an 8-bit-wide memory will have 8 million distinct addresses.

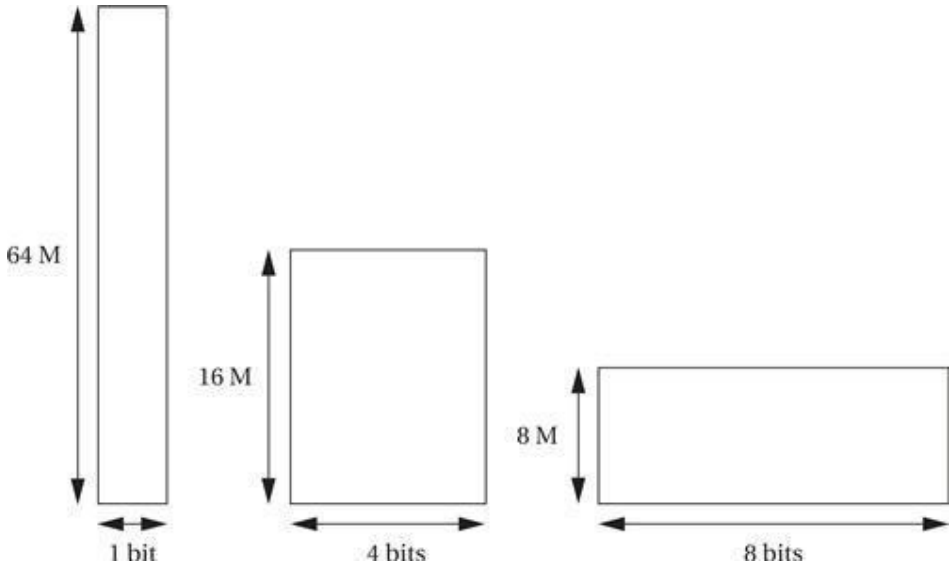


Figure 2.16 Memory aspect ratios.

Components for Embedded Programs:

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.

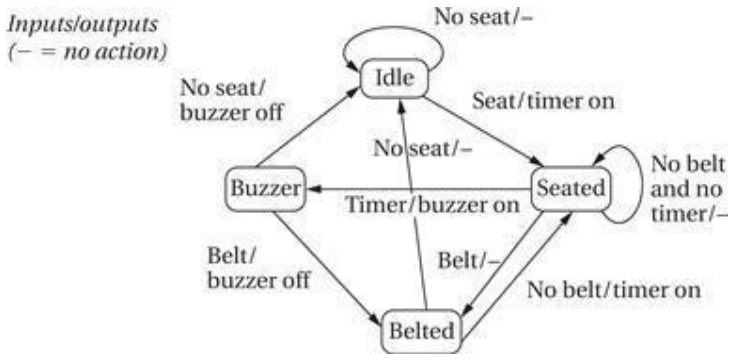
State Machines:

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 behavior.

Programming Example 5.1 shows how to write a finite-state machine in a high-level programming language.

- The behavior we want to implement is a simple **seat belt controller**.
- The controller's job is to turn on a buzzer if a person sits in a seat and does not fasten the seat belt within a fixed amount of time.
- This system has three inputs and one output. The inputs are a sensor for the seat to know when a person has sat down, a seat belt sensor that tells when the belt is fastened, and a timer that goes off when the required time interval has elapsed.

- The output is the buzzer. Appearing below is a state diagram that describes the seat belt controller's behavior.



- The idle state is in force when there is no person in the seat.
- When the person sits down, the machine goes into the seated state and turns on the timer.
- If the timer goes off before the seat belt is fastened, the machine goes into the buzzer state.
- If the seat belt goes on first, it enters the belted state.
- When the person leaves the seat, the machine goes back to idle.

We will use a variable named state to hold the current state of the machine and a switch statement to determine what action to take in each state. Here is the code:

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
        switch(state) { /* check the current state */
        case IDLE:
        if (seat){ state = SEATED; timer_on = TRUE; }
        /* default case is self-loop */
        break;
case SEATED:
        if (belt) state = BELTED; /* won't hear the buzzer */
        else if (timer) state = BUZZER; /* didn't put on belt in time */
        /* default case is self-loop */
 break;
        case BELTED:
        if (!seat) state = IDLE; /* person left */
        else if (!belt) state = SEATED; /* person still in seat */
        break;
case BUZZER:
        if (belt) state = BELTED; /* belt is on---turn off buzzer */
        else if (!seat) state = IDLE; /* no one in seat--turn off buzzer */
        break;
    }
```

Circular Buffers and Stream-Oriented Programming

Data stream style:

The data stream style makes sense for data that comes in regularly and must be processed on the fly. The FIR filter is a classic example of stream-oriented processing. 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.

Circular buffer:

The **circular buffer** is a data structure that lets us handle streaming data in an efficient way. Figure 2.17 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. Because the size of the window does not change, we can use a fixed-size buffer to hold the current data.

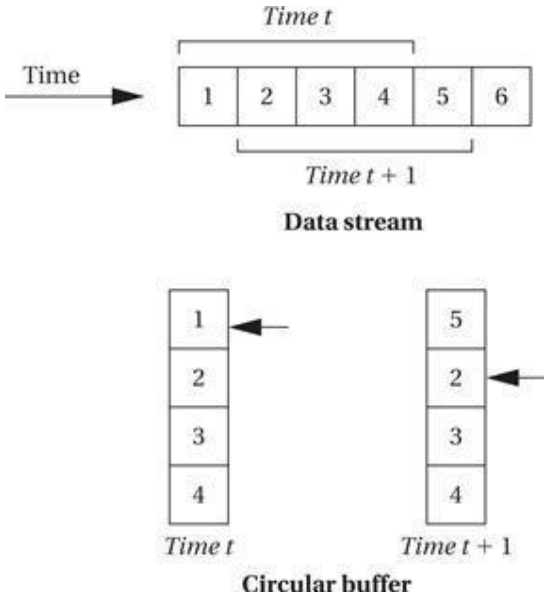


Figure 2.17A circular buffer.

Programming Example:A Circular Buffer in C

Once we build a circular buffer, we can use it in a variety of ways. We will use an array as the buffer:

```
#define CMAX 6 /* filter order */

int circ[CMAX]; /* circular buffer */

int pos; /* position of current sample */
```

The variable pos holds the position of the current sample. As we add new values to the buffer this variable moves. Here is the function that adds a new value to the buffer:

```
void circ_update(int xnew) {

     /* add the new sample and push off the oldest one */

     /* compute the new head value with wraparound; the pos pointer moves from
0 to CMAX-1 */

     pos = ((pos == CMAX-1) ? 0 : (pos+1));

     /* insert the new value at the new head */

     circ[pos] = xnew;

     }
```

We can now write an initialization function. It sets the buffer values to zero. More important, it sets pos to the initial value. For ease of debugging, we want the first data element to go into circ[0]. To do this, we set pos to the end of the array so that it is set to zero before the first element is added:

```
void circ_init(){

     int i;

     for (i=0; i<CMAX; i++) /* set values to 0 */

        circ[i] = 0;

     pos=CMAX-1; /* start at tail so first element will be at 0 */

     }
```

We can also make use of a function to get the i th value of the buffer. This function has to

translate the index in temporal order—zero being the newest value—to its position in the array:

```
int circ_get(int i) {
    /* get the ith value from the circular buffer */
  int ii;
    /* compute the buffer position */
  ii = (pos - i) % CMAX;
  /* return the value */
    return circ[ii];
    }
```

QUEUES

Queues are also used in signal processing and event processing. Queues are used whenever data may arrive and depart at somewhat unpredictable times or when variable amounts of data may arrive. A queue is often referred to as an **elastic buffer.**

Models of Programs

In this section, we develop models for programs that are more general than source code. Why not use the source code directly? First, there are many different types of source code—assembly languages, C code, and so on—but we can use a single model to describe all of them.

Our fundamental model for programs is the control/data flow graph (CDFG). (We can also model hardware behavior with the CDFG.) As the name implies, the CDFG has constructs that model both data operations (arithmetic and other computations) and control operations (conditionals).

Part of the power of the CDFG comes from its combination of control and data constructs. To understand the CDFG, we start with pure data descriptions and then extend the model to control.

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 2.18 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.

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.

```
w  = a + b;
x1 = a - c;
y  = x1 + d;
x2 = a + c;
z  = y + e;
```

The basic block in single-assignment form.

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. The data flow graph for our single-assignment code is shown in Figure 2.18.

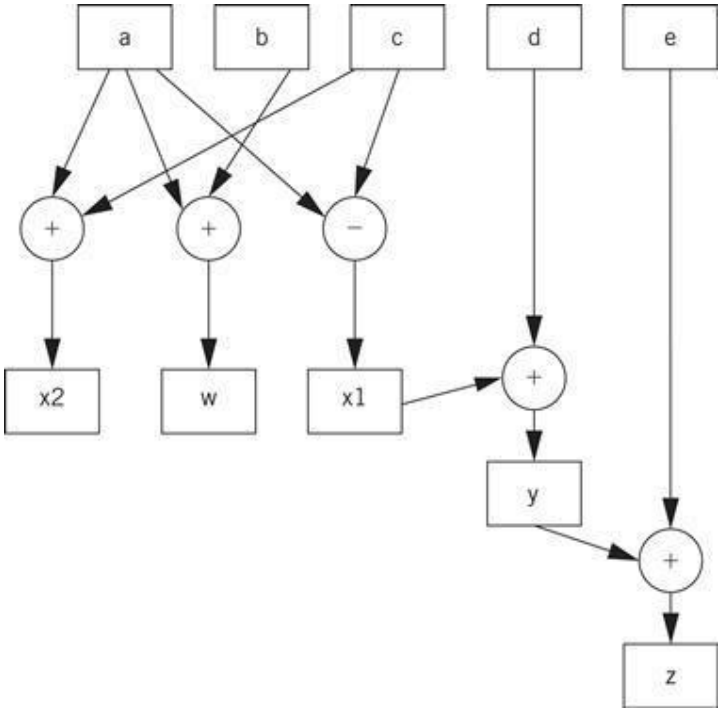


Figure 2.18. An extended data flow graph for our sample basic block.

The data flow graph is generally drawn in the form shown in Figure 2.19. Here, the variables are not explicitly represented by nodes. Instead, the edges are labeled with the variables they represent.

As a result, a variable can be represented by more than one edge. However, the edges are directed and all the edges for a variable must come from a single source. We use this form for its simplicity and compactness.

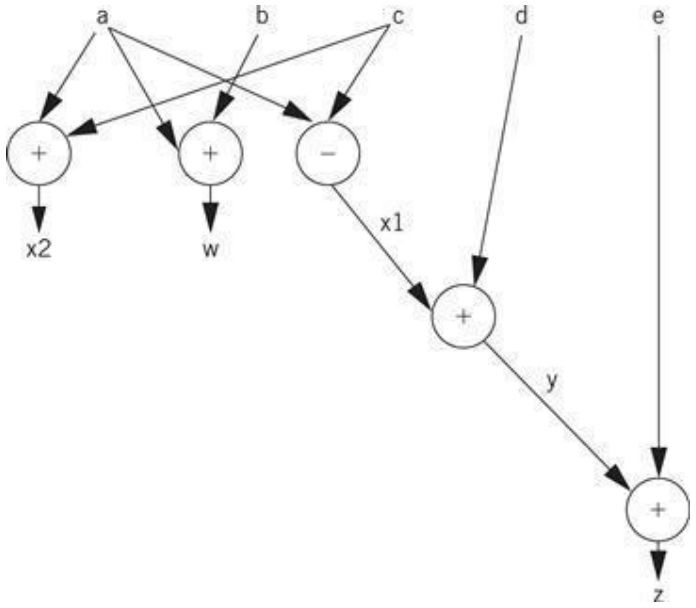


Figure 2.19. Standard data flow graph for our sample basic block.

CONTROL/DATA FLOW GRAPHS

A CDFG uses a data flow graph as an element, adding constructs to describe control. In a basic CDFG, we have two types of nodes: **decision nodes** and **data flow nodes**.

A data flow node encapsulates a complete data flow graph to represent a basic block. We can use one type of decision node to describe all the types of control in a sequential program. (The jump/branch is, after all, the way we implement all those high-level control constructs.)

Figure 2.20 shows a bit of C code with control constructs and the CDFG constructed from it. The rectangular nodes in the graph represent the basic blocks. The basic blocks in the C code have been represented by function calls for simplicity. The diamond-shaped nodes represent the conditionals.

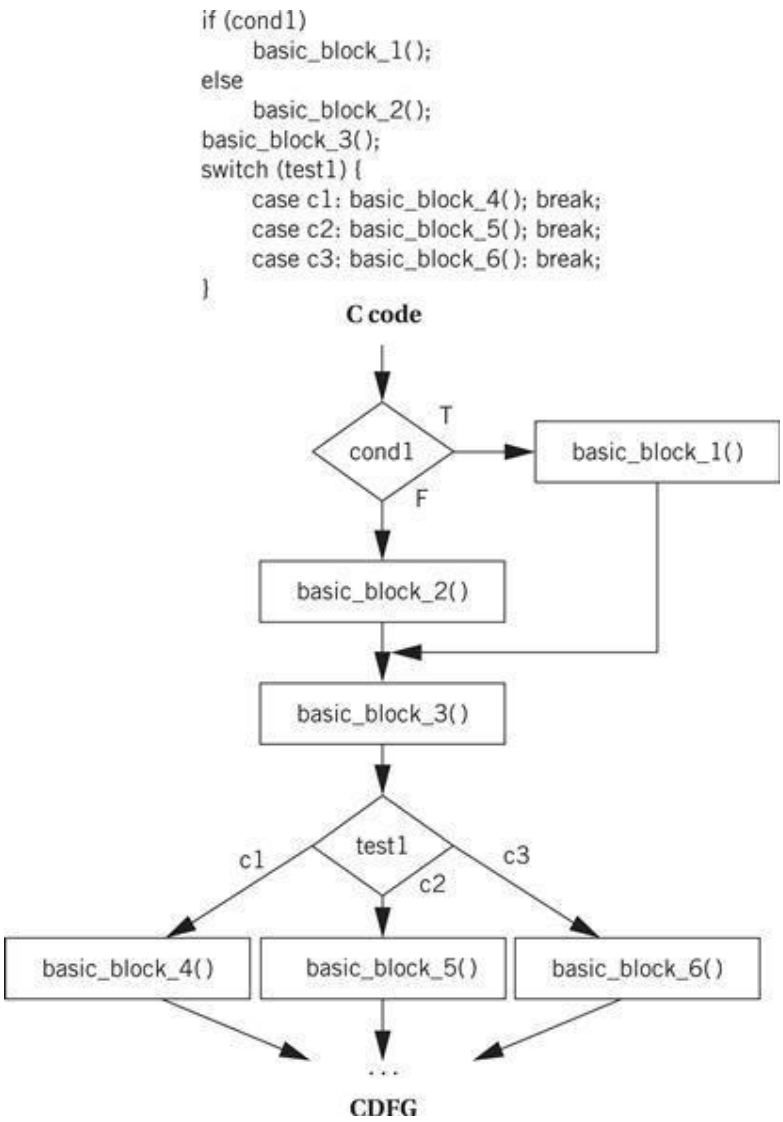


Figure 2.20 C code and its CDFG.

Building a CDFG for a while loop is straightforward, as shown in Figure 2.21. The while loop consists of both a test and a loop body, each of which we know how to represent in a CDFG. We can represent for loops by remembering that, in C, a for loop is defined in terms of a while loop.

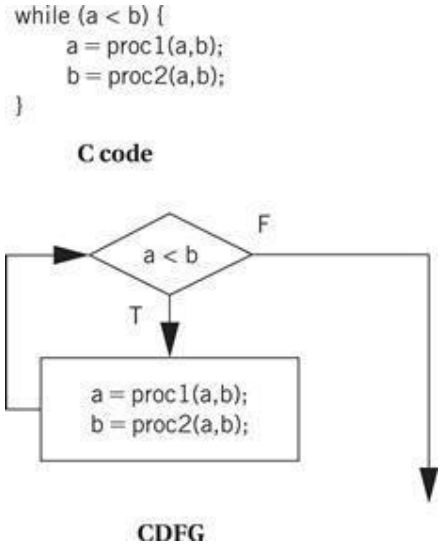


Figure 2.21 A while loop and its CDFG.

Assembly, Linking, and Loading

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.

Program generation work flow:

Figure 2.22 highlights the role of assemblers and linkers in the compilation process.. 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.

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, because 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.**

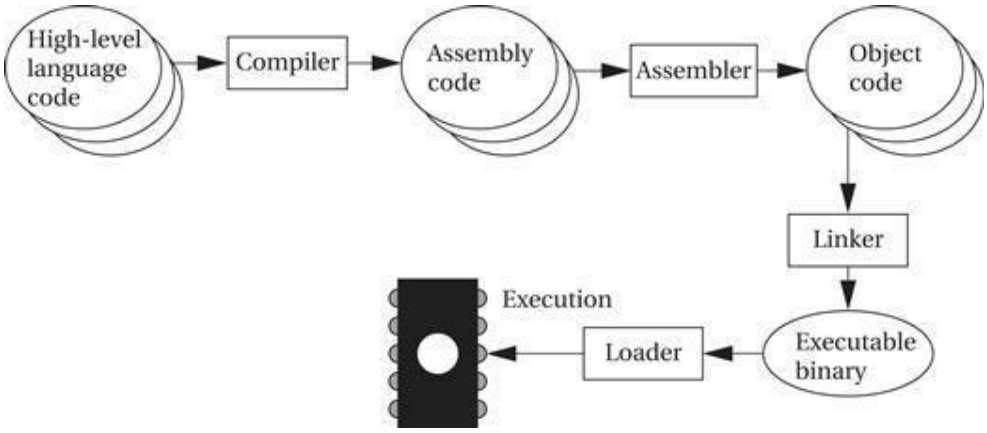


Figure 2.22 Program generation from compilation through loading.

Absolute and relative addresses

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.**

Most assemblers therefore allow us to use **relative addresses** by specifying at the start of the file that the origin of the assembly language module is to be computed later. Addresses within the module are then computed relative to the start of the module. The linker is then responsible for translating relative addresses into 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.

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:

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.

Symbol table

As shown in Figure 2.23, 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.

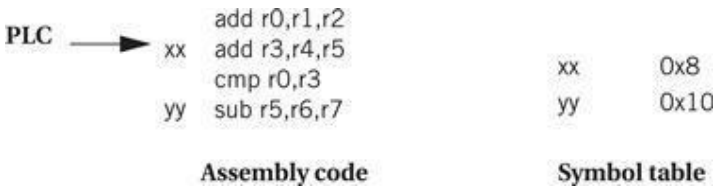
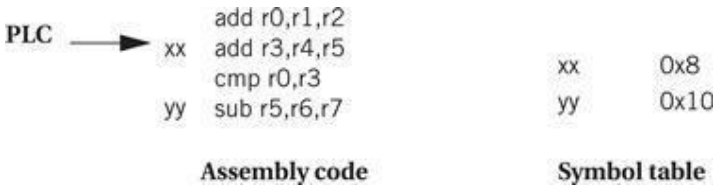


Figure 2.23 Symbol table processing during assembly.

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 (because ARM instructions are four bytes long, the PLC would be incremented by four) and looks at the next instruction.

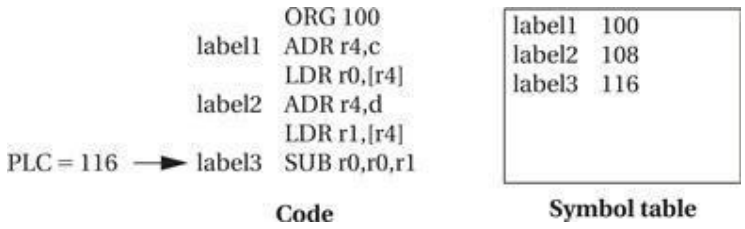
If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The value of the label is equal to the current value of the PLC.

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.



Symbol table processing during assembly.

Generating a Symbol Table



Object code formats:

The assembler produces an object file that describes the instructions and data in binary format. A commonly used object file format, originally developed for Unix but now used in other environments as well, is known as COFF (common object file format). The object file must describe the instructions, data, and any addressing information and also usually carries along the symbol table for later use in debugging.

LINKING:

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 Figure2.24. 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.

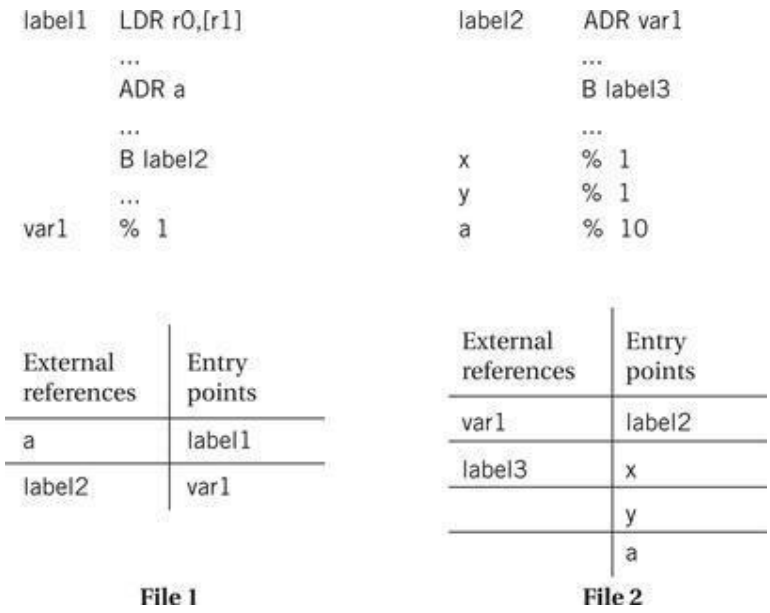


Figure2.24. External references and entry points.

Linking process:

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.

Compilation Techniques

Even though we don't write our own assembly code much of the time, we still care about the characteristics of the code our compiler generates: its speed, its size, and its power consumption. Understanding how a compiler works will help us write code and direct the compiler to get the assembly language implementation we want.

THE COMPILATION PROCESS

It is useful to understand how a high-level language program is translated into instructions: interrupt handling instructions, placement of data and instructions in memory, etc

We can summarize the compilation process with a formula:

$$compilation = translation + optimization$$

The high-level language program is translated into the lower-level form of instructions; optimizations try to generate better instruction sequences than would be possible if the brute force technique of independently translating source code statements were used.

The compilation process is outlined in Figure 2.25. Compilation begins with high-level language code such as C or 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.

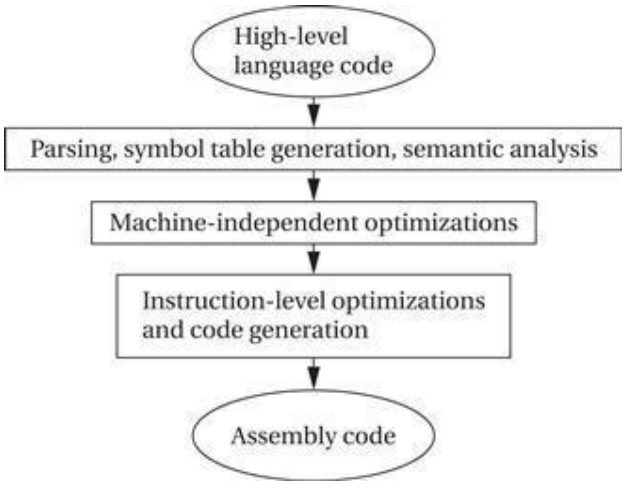


Figure 2.25 The compilation process.

Simplifying arithmetic expressions is one example of **a machine-independent optimization**.

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.

BASIC COMPILATION METHODS

Statement translation:

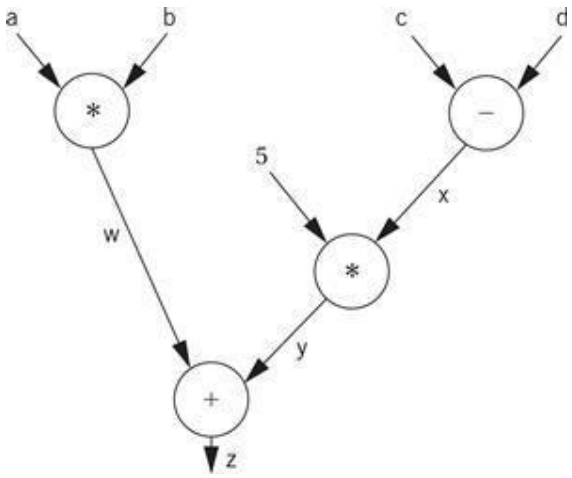
Let's first consider how to translate an expression. A large amount of the code in a typical application consists of arithmetic and logical expressions. Understanding how to compile a single expression, as described in the next example, is a good first step in understanding the entire compilation process.

Compiling an Arithmetic Expression

Consider this arithmetic expression:

```
x = a*b + 5*(c − d)
```

The code for the expression can be built by walking the data flow graph. Here is the data flow graph for the expression.



Here is the resulting ARM code:

```
; operator 1 (+)

ADR r4,a      ; get address for a

MOV r1,[r4]     ; load a
```

```
ADR r4,b      ; get address for b
MOV r2,[r4]     ; load b
MUL r3,r1,r2   ; put w into r3
; operator 2 (-)
ADR r4,c      ; get address for c
MOV r4,[r4]   ; load c
ADR r4,d     ; get address for d
MOV r5,[r4]   ; load d
SUB r6,r4,r5  ; put z into r6
; operator 3 (*)
MUL r7,r6,#5  ; operator 3, puts y into r7
; operator 4 (+)
ADD r8,r7,r3  ; operator 4, puts x into r8
; assign to x
ADR r1,x
```

STR r8,[r1] ; assigns to x location

One obvious optimization is to reuse a register whose value is no longer needed. In the case of the intermediate values w, y, and z, we know that they cannot be used after the end of the expression (e.g., in another expression) because they have no name in the C program.

However, the final result z may in fact be used in a C assignment and the value reused later in the program. In this case we would need to know when the register is no longer needed to determine its best use.

We also need to be able to translate control structures. Because conditionals are controlled by expressions, the code generation techniques of the last example can be used for those expressions, leaving us with the task of generating code for the flow of control itself. Figure 2.26 shows a simple example of changing flow of control in C—an if statement, in which the condition controls whether the true or false branch of the if is taken. Figure 2.26 also shows the control flow diagram for the if statement.

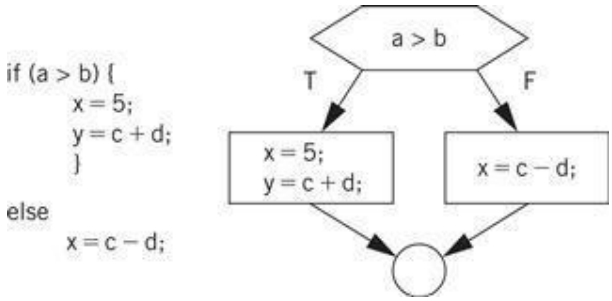
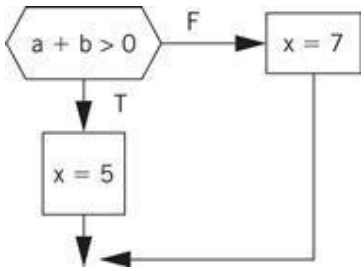


Figure 2.26 Flow of control in C and control flow diagrams.

Example : Generating Code for a Conditional

Consider this C statement: if (a + b > 0)
x = 5;
else x = 7;

The CDFG for the statement appears below.




```
 ADR r5,a    ; get address for a
   LDR r1,[r5]    ; load a
   ADR r5,b ; get address for b
   LDR r2,b    ; load b
   ADD r3,r1,r2
   BLE label3 ; true condition falls through branch
; true case
   LDR r3,#5    ; load constant
   ADR r5,x
   STR r3, [r5] ; store value into x
   B stmtend    ; done with the true case
; false case
```

```
label3   LDR r3,#7    ; load constant

   ADR r5,x        ; get address of x

   STR r3,[r5]      ; store value into x

stmtend   …
```

2.9.3 COMPILER OPTIMIZATIONS

Basic compilation techniques can generate inefficient code. Compilers use a wide range of algorithms to optimize the code they generate.

Loop transformations :

Loops are important program structures—although they are compactly described in the source code, they often use a large fraction of the computation time. Many techniques have been designed to optimize loops.

A simple but useful transformation is known as **loop unrolling**, illustrated in the next example. Loop unrolling is important because it helps expose parallelism that can be used by later stages of the compiler.

Example: Loop Unrolling

Here is a simple C loop:

```
for (i = 0; i < N; i++) {

   a[i]=b[i]*c[i];

}
```

If we let N = 4, then we can substitute this straight-line code for the loop:

```
a[0] = b[0]*c[0];

a[1] = b[1]*c[1];

a[2] = b[2]*c[2];

a[3] = b[3]*c[3];
```

Rather than unroll the above loop four times, we could unroll it twice. Unrolling produces this code:

```
for (i = 0; i < 2; i++) {
```

```
a[i*2] = b[i*2]*c[i*2];

a[i*2 + 1] = b[i*2 + 1]*c[i*2 + 1];

}
```

Loop fusion combines two or more loops into a single loop. For this transformation to be legal, two conditions must be satisfied. First, the loops must iterate over the same values. Second, the loop bodies must not have dependencies that would be violated if they are executed together

Loop distribution is the opposite of loop fusion that is, decomposing a single loop into multiple loops.

Loop tiling breaks up a loop into a set of nested loops, with each inner loop performing the operations on a subset of the data.

Dead code elimination:

Dead code is code that can never be executed. Dead code can be generated by programmers, either inadvertently or purposefully. Dead code can also be generated by compilers. Dead code can be identified by **reachability analysis**—finding the other statements or instructions from which it can be reached.

Register allocation:

Register allocation is a very important compilation phase. Given a block of code, we want to choose assignments of variables (both declared and temporary) to registers to minimize the total number of required registers.

Example: Register Allocation

Consider this C code:

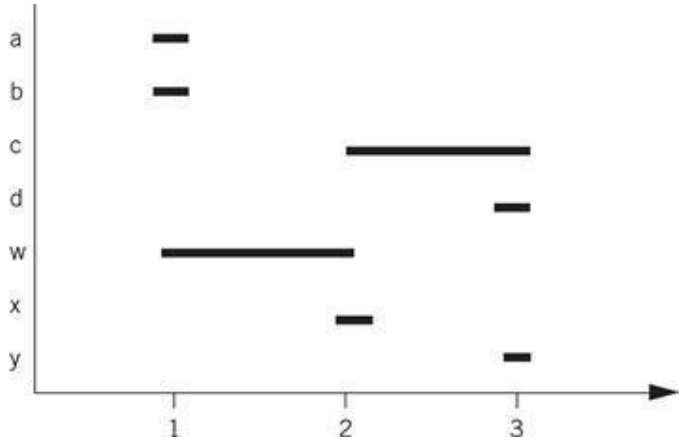
```
w = a + b; /* statement 1 */

x = c + w; /* statement 2 */

y = c + d; /* statement 3 */
```

A naive register allocation, assigning each variable to a separate register, would require seven registers for the seven variables in the above code. However, we can do much better by reusing a register once the value stored in the register is no longer needed.

we can draw a lifetime graph that shows the statements on which each statement is used. Here is a lifetime graph in which the x axis is the statement number in the C code and the y axis shows the variables.



By reusing registers once their current values are no longer needed, we can write code that requires no more than four registers. Here is one register assignment:

a	r0
b	r1
c	r2
d	r0
w	r3
x	r0
y	r3

Here is the ARM assembly code that uses the above register assignment:

```
LDR r0,[p_a]   ; load a into r0 using pointer to a (p_a)

LDR r1,[p_b]   ; load b into r1

ADD r3,r0,r1   ; compute a + b

STR r3,[p_w]   ; w = a + b

LDR r2,[p_c]   ; load c into r2

ADD r0,r2,r3   ; compute c + w, reusing r0 for x

STR r0,[p_x]   ; x = c + w

LDR r0,[p_d]   ; load d into r0

ADD r3,r2,r0   ; compute c + d, reusing r3 for y

STR r3,[p_y]   ; y = c + d
```

Scheduling:

We have some freedom to choose the order in which operations will be performed. We can use this to our advantage—for example, we may be able to improve the register allocation by changing the order in which operations are performed, thereby changing the lifetimes of the variables.

We can keep track of CPU resources during instruction scheduling using a **reservation table.**

As illustrated in Figure rows in the table represent instruction execution time slots and columns represent resources that must be scheduled. Before scheduling an instruction to be executed at a particular time, we check the reservation table to determine whether all resources needed by the instruction are available at that time.

Time	Resource A	Resource B
t	X	
t + 1	X	X
t + 2	X	
t + 3		X

A reservation table for instruction scheduling.

We can also schedule instructions to maximize performance. As we know, when an instruction that takes more cycles than normal to finish is in the pipeline, pipeline bubbles appear that reduce performance.

Software pipelining is a technique for reordering instructions across several loop iterations to reduce pipeline bubbles. Some instructions take several cycles to complete; if the value produced by one of these instructions is needed by other instructions in the loop iteration, then they must wait for that value to be produced.

Program-Level Performance Analysis

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.

As illustrated in Figure 2.27, 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.

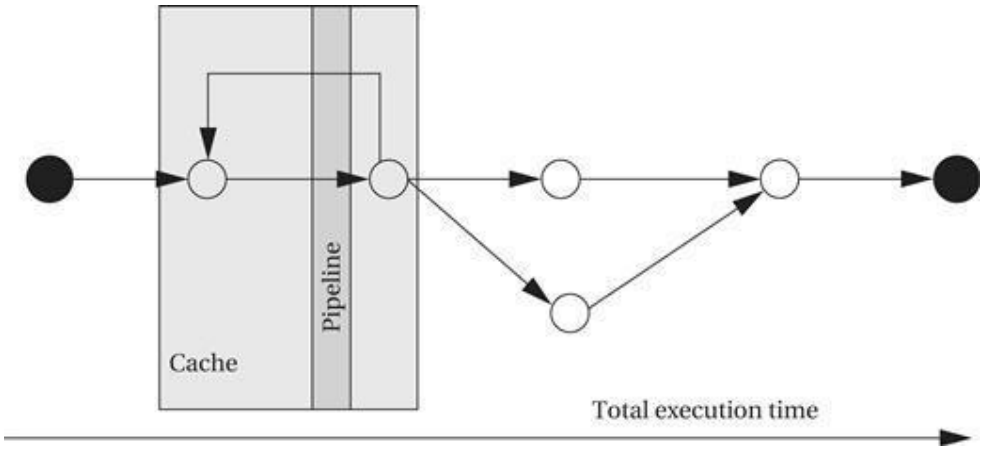


Figure 2.27 Execution time is a global property of a program.

Difficulties for determining execution time of programs.

- 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.

Measuring execution speed:

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.
- 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 analyzer's 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

ELEMENTS OF PROGRAM PERFORMANCE

Program execution time can be seen as

$$execution\,time = program\,path + instruction\,timing$$

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 behavior, and caching.

Instruction timing:

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, which means we need only count the instructions and multiply by the per-instruction execution time to obtain 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*
- *Execution times of instructions are not independent.*
- *The execution time of an instruction may depend on operand values.*

MEASUREMENT-DRIVEN PERFORMANCE ANALYSIS

The most direct way to determine the execution time of a program is by measuring it. This approach is appealing but it does have some drawbacks. First, in order to cause the program to execute its worst-case execution path, we have to provide the proper inputs to it.

Determining the set of inputs that will guarantee the worst-case execution path is infeasible. Furthermore, in order to measure the program's performance on a particular type of CPU, we need the CPU or its simulator.

Despite these problems, measurement is the most commonly used way to determine the execution time of embedded software. Worst-case execution time analysis algorithms have been used successfully in some areas, such as flight control software, but many system design projects determine the execution time of their programs by measurement.

Program traces:

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** . Traces can be valuable for other purposes, such as analyzing the cache behavior of the program.

Measurement issues:

Perhaps the biggest problem in measuring program performance is figuring out a useful set of inputs to give 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.

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.

Profiling:

Profiling is a simple method for analyzing software performance. A profiler does not measure execution time—instead, it counts the number of times that procedures or basic blocks in the program are executed.

There are two major ways to profile a program: we can modify the executable program by adding instructions that increment a location every time the program passes that point in the program;

or we can sample the program counter during execution and keep track of the distribution of PC values. Profiling adds relatively little overhead to the program and it gives us some useful information about where the program spends most of its time.

Physical performance measurement:

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.

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.

Software Performance Optimization

Techniques for optimizing software performance, including both loop and cache optimizations.

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. A simple example of code motion is also common. Consider this loop:

```
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 2.28

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.

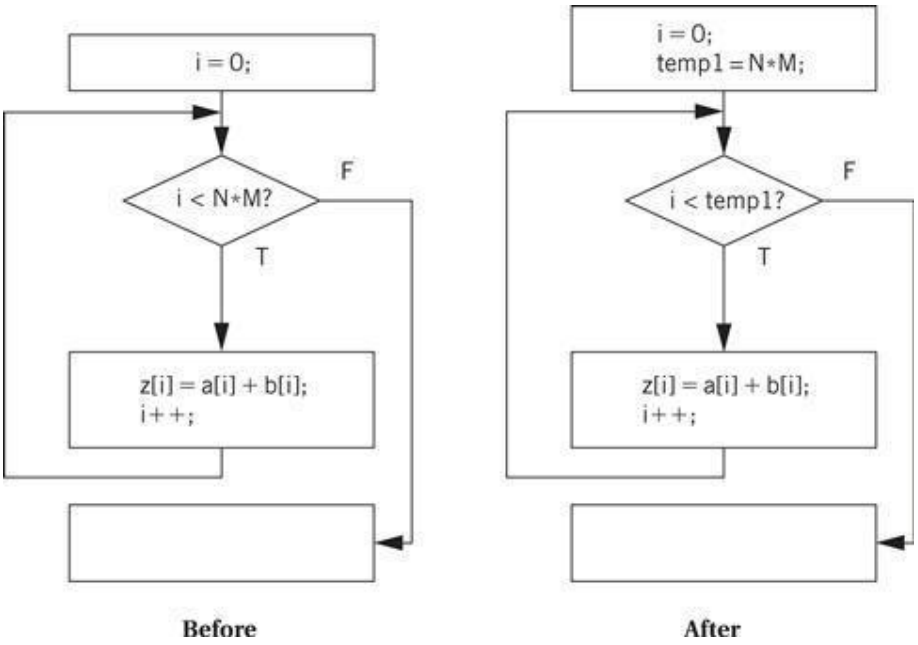


Figure 2.28 Code motion in a loop.

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];
```

The compiler uses induction variables to help it address the arrays. Let us rewrite the loop in C using induction variables and pointers.

```
for (i = 0; i < N; i++)
```

```
for (j = 0; j < M; j++) {
    zbinduct = i*M + j;
    *(zptr + zbinduct) = *(bptr + zbinduct);
    }
```

Strength reduction helps us reduce the cost of a loop iteration. Consider this assignment:

y = x * 2;

In integer arithmetic, we can use a left shift rather than a multiplication by 2. If the shift is faster than the multiply, we probably want to perform the substitution.

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.

In this section we concentrate on the analysis of loop nests for cache performance.

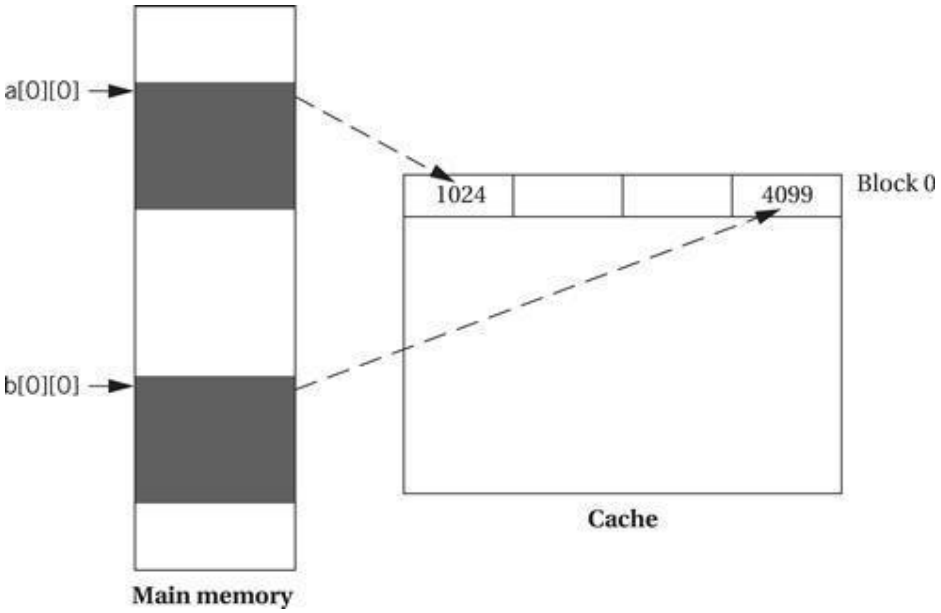
Programming Example : Data Realignment and Array Padding.

Assume we want to optimize the cache behavior of the following code:

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

Let us also assume that the a and b arrays are sized with M at 265 and N at 4 and a 256-line, four-way set-associative cache with four words per line.

Assume that the starting location for a[] is 1024 and the starting location for b[] is 4099. Although a[0][0] and b[0][0] do not map to the same word in the cache, they do map to the same block.



As a result, we see the following scenario in execution:

• The access to a[0][0] brings in the first four words of a[].

• The access to b[0][0] replaces a[0][0] through a[0][3] with b[0][3] and the contents of the three locations before b[].

• When a[0][1] is accessed, the same cache line is again replaced with the first four elements of a[].

Once the a[0][1] access brings that line into the cache, it remains there for the a[0][2] and a[0][3] accesses because the b[] accesses are now on the next line. However, the scenario repeats itself at a[1][0] and every four iterations of the cache.

One way to eliminate the cache conflicts is to move one of the arrays. We do not have to move it far. If we move b's start to 4100, we eliminate the cache conflicts.

Program-Level Energy and Power Analysis and Optimization:

Power consumption is a particularly important design metric for battery-powered systems because the battery has a very limited lifetime.

How much control do we have over power consumption? Ultimately, we must consume the energy required to perform necessary computations.

However, there are opportunities for saving power:

- We may be able to replace the algorithms with others that do things in clever ways that consume less power.
- Memory accesses are a major component of power consumption in many applications. By optimizing memory accesses we may be able to significantly reduce power.
- We may be able to turn off parts of the system—such as subsystems of the CPU, chips in the system, and so on—when we don't need them in order to save power.

The first step in optimizing a program's energy consumption is knowing how much energy the program consumes. It is possible to measure power consumption for an instruction or a small code fragment.

The technique, illustrated in Figure 2.29, executes the code under test over and over in a loop. By measuring the current flowing into the CPU, we are measuring the power consumption of the complete loop, including both the body and other code.

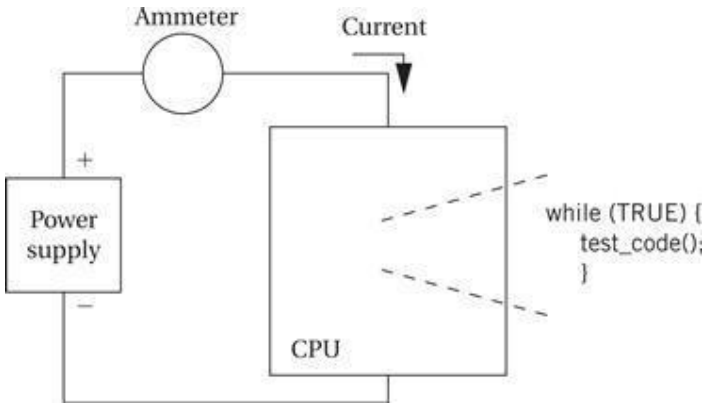


Figure 2.29, Measuring energy consumption for a piece of code.

Measuring energy consumption for a piece of code.

Several factors contribute to the energy consumption of the program:

• Energy consumption varies somewhat from instruction to instruction.

• The sequence of instructions has some influence.

• The opcode and the locations of the operands also matter.

Energy optimization:

How can we optimize a program for low power consumption? The best overall advice is that

high performance = low power.

Generally speaking, making the program run faster also reduces energy consumption. A few optimizations mentioned previously for performance are also often useful for improving energy consumption:

A few optimizations mentioned previously for performance are also often useful for improving energy consumption:

- Try to use registers efficiently.
- Analyze cache behavior to find major cache conflicts. Restructure the code to eliminate as many of these as you can:
- For instruction conflicts, if the offending code segment is small, try to rewrite the segment to make it as small as possible so that it better fits into the cache.

    - For scalar data conflicts, move the data values to different locations to reduce conflicts.
    - For array data conflicts, consider either moving the arrays or changing your array access patterns to reduce conflicts.
    - Software pipelining reduces pipeline stalls, thereby reducing the average energy per instruction.

EMBEDDED SYSTEMS

UNIT III

PROCESSES AND OPERATING SYSTEMS

MULTIPLE TASKS AND PROCESSES:

Most embedded systems require functionality and timing that is too complex to embody in a single program. We break the system into multiple tasks in order to manage when things happen. In this section we will develop the basic abstractions that will be manipulated by the RTOS to build multirate systems.

Tasks and Processes

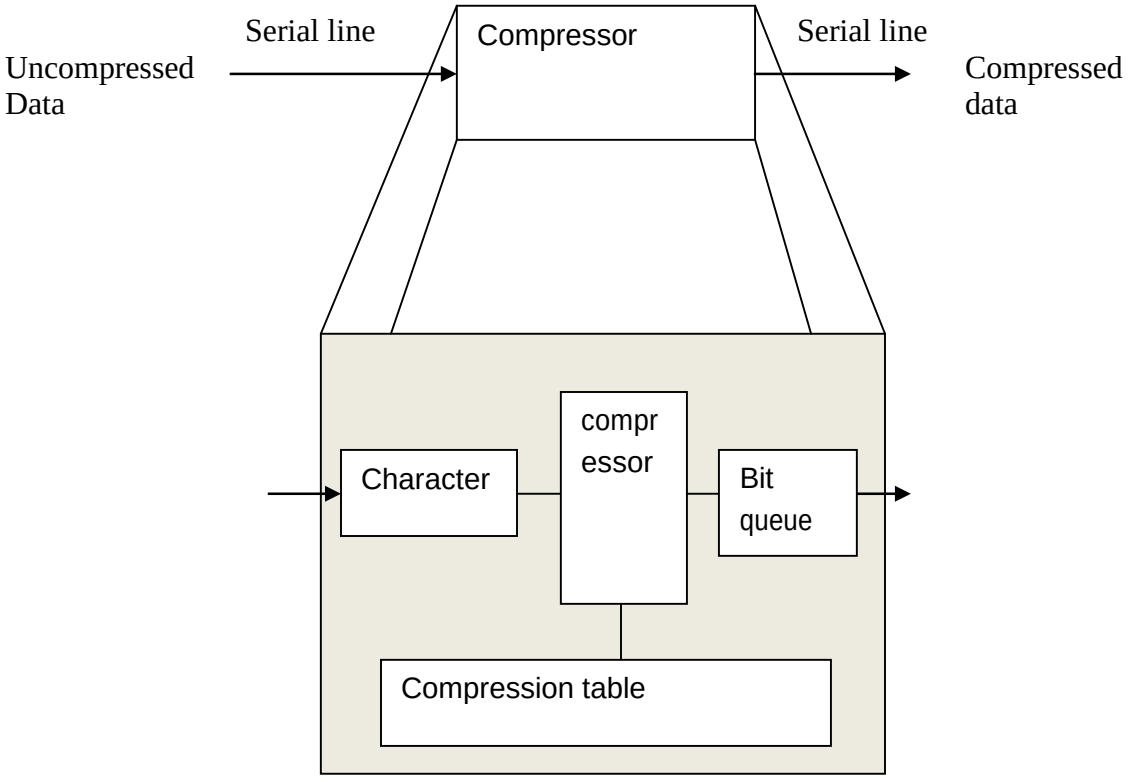
Many (if not most) embedded computing systems do more than one thing—that is, the environment can cause mode changes that in turn cause the embedded system to behave quite differently. For example, when designing a telephone answering machine, we can define recording a phone call and operating the user's 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 hasits 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***.

To understand why the separation of an application into tasks may be reflected in the program structure, consider how we would build a stand-alone compression unit based on the compression algorithm. 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. We do know, however, that the button will be depressed at a much lower rate than characters will be received, since it is not physically possible for a person to repeatedly depress a button at even slow serial line rates. Keeping up with the input and output data while checking on the button can introduce some very complex control code into the program. Sampling the button's state too slowly can cause the machine to miss a button depression entirely, but sampling it too frequently and duplicating a data value can cause the machine to incorrectly compress data. One solution is to introduce a counter into the main compression loop, so that a subroutine to check the input button is called once every *n* times the compression loop is executed. But this solution does not work when either the compression loop or the button-handling routine has highly variable execution times—if the execution time of either varies significantly, it will cause the other to execute later than expected, possibly causing data to be lost. We need to be able to keep track of these two different tasks separately, applying different timing requirements to each. This is the sort of control that processes allow. The above two examples illustrate how requirements on timing and execution rate can create major problems in programming. When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution become very complex very quickly. Worse, such complex control is usually quite difficult to verify for either functional or timing properties.



CONTEXT SWITCHING

In computing, a **context switch** is the process of storing and restoring the state (context) of a process so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system. What constitutes the context is determined by the processor and the operating system.

Context switches are usually computationally intensive, and much of the design of operating systems is to optimize the use of context switches. Switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and lists etc.

A context switch can mean a register context switch, a task context switch, a stack frame switch, a thread context switch, or a process context switch.

Multitasking

Most commonly, within some scheduling scheme, one process needs to be switched out of the CPU so another process can run. This context switch can be triggered by the process making itself unrunnable, such as by waiting for an I/O or synchronization operation to complete. On a pre-emptive multitasking system, the scheduler may also switch out processes which are still runnable. To prevent other processes from being starved of CPU time, preemptive schedulers often configure a timer interrupt to fire when a process exceeds its time slice. This interrupt ensures that the scheduler will gain control to perform a context switch.

Interrupt handling

Modern architectures are interrupt driven. This means that if the CPU requests data from a disk, for example, it does not need to busy-wait until the read is over; it can issue the request and continue with some other execution. When the read is over, the CPU can be *interrupted* and presented with the read. For interrupts, a program called an *interrupt handler* is installed, and it is the interrupt handler that handles the interrupt from the disk.

When an interrupt occurs, the hardware automatically switches a part of the context (at least enough to allow the handler to return to the interrupted code). The handler may save additional context, depending on details of the particular hardware and software designs. Often only a minimal part of the context is changed in order to minimize the amount of time spent handling the interrupt. The kernel does not spawn or schedule a special process to handle interrupts, but instead the handler executes in the (often partial) context established at the beginning of interrupt handling. Once interrupt servicing is complete, the context in effect before the interrupt occurred is restored so that the interrupted process can resume execution in its proper state.

User and kernel mode switching

When a transition between user mode and kernel mode is required in an operating system, a context switch is not necessary; a mode transition is *not* by itself a context switch. However, depending on the operating system, a context switch may also take place at this time.

Steps

In a switch, the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue.

The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. This data is usually stored in a data structure called a process control block (PCB), or switchframe.

In order to switch processes, the PCB for the first process must be created and saved. The PCBs are sometimes stored upon a per-process stack in kernel memory (as opposed to the user-mode call stack), or there may be some specific operating system defined data structure for this information.

Since the operating system has effectively suspended the execution of the first process, it can now load the PCB and context of the second process. In doing so, the program counter from the PCB is loaded, and thus execution can continue in the new process. New processes are chosen from a queue or queues. Process and thread priority can influence which process continues execution, with processes of the highest priority checked first for ready threads to execute.

Software vs hardware context switching

Context switching can be performed primarily by software or hardware. Some processors, like the Intel 80386 and its successors,[1] have hardware support for context switches, by making use of a special data segment designated the Task State Segment or TSS. A task switch can be explicitly triggered with a CALL or JMP instruction targeted at a TSS descriptor in the global descriptor table. It can occur implicitly when an interrupt or exception is triggered if there's a task gate in the interrupt descriptor table. When a task switch occurs the CPU can automatically load the new state from the TSS. As with other tasks performed in hardware, one would expect this to be rather fast; however, mainstream operating systems, including Windows and Linux,[2] do not use this feature.

This is due to mainly two reasons:

1. hardware context switching does not save all the registers (only general purpose registers, not floating point registers — although the TS bit is automatically turned on in the CR0 control register, resulting in a fault when executing floating point instructions and giving the OS the opportunity to save and restore the floating point state as needed).
2. associated performance issues, e.g., software context switching can be selective and store only those registers that need storing, whereas hardware context switching stores nearly all registers whether they are required or not.

SCHEDULING POLICY

In computer science, **scheduling** is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance and share system resources effectively or achieve a target quality of service. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (executing more than one process at a time) and multiplexing (transmit multiple data streams simultaneously across a single physical channel).

The scheduler is concerned mainly with:

- Throughput - The total number of processes that complete their execution per time unit.
- Latency, specifically:
  - Turnaround time - total time between submission of a process and its completion.
  - Response time - amount of time it takes from when a request was submitted until the first response is produced.
- Fairness - Equal CPU time to each process (or more generally appropriate times according to each process' priority and workload).
- Waiting Time - The time the process remains in the ready queue.

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the above mentioned concerns depending upon the user's needs and objectives.

In real-time environments, such as embedded systems for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable. Scheduled tasks can also be distributed to remote devices across a network and managed through an administrative back end.

Types of operating system schedulers

Operating systems may feature up to three distinct types of scheduler, a *long-term scheduler* (also known as an admission scheduler or high-level scheduler), a *mid-term or medium-term scheduler* and a *short-term scheduler*. The names suggest the relative frequency with which these functions are performed. The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run.

Process scheduler

Long-term scheduling

The long-term, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue (in the Main Memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler . Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - i.e.: whether a high or low amount of processes are to be executed concurrently, and how the split between I/O intensive and CPU intensive processes is to be handled. The long term scheduler is responsible for controlling the

degree of multiprogramming . In modern operating systems, this is used to make sure that real time processes get enough CPU time to finish their tasks. Without proper real time scheduling, modern GUIs would seem sluggish.

Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers and render farms. In these cases, special purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system..

Medium term scheduling

Scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource. [Stallings, 396] [Stallings, 370]

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the medium-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded". [Stallings, 394]

Short-term scheduling

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes are to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU.

A preemptive scheduler relies upon a programmable interval timer which invokes an interrupt handler that runs in kernel mode and implements the scheduling function.

Dispatcher

Another component that is involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program.

Dispatcher analyses the values from Program counter and fetches instructions, loads data into registers.

The dispatcher should be as fast as possible, since it is invoked during every process switch. During the context switches, the processor is idle for a fraction of time. Hence, unnecessary context switches should be avoided. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**. [Galvin, 155].

Network scheduler

Main article: Network scheduler

Disk scheduler

Main article: I/O scheduling

Job scheduler

Main article: Job scheduler

Examples are cron, at, systemd.

Scheduling disciplines

Scheduling disciplines are algorithms used for distributing resources among parties which simultaneously and asynchronously request them. Scheduling disciplines are used in routers (to handle packet traffic) as well as in operating systems (to share CPU time among both threads and processes), disk drives (I/O scheduling), printers (print spooler), most embedded systems, etc.

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources. Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms. In this section, we introduce several of them.

In packet-switched computer networks and other statistical multiplexing, the notion of a **scheduling algorithm** is used as an alternative to first-come first-served queuing of data packets.

The simplest best-effort scheduling algorithms are round-robin, fair queuing (a max-min fair scheduling algorithm), proportionally fair scheduling and maximum throughput. If differentiated or guaranteed quality of service is offered, as opposed to best-effort communication, weighted fair queuing may be utilized.

In advanced packet radio wireless networks such as HSDPA (High-Speed Downlink Packet Access) 3.5G cellular system, **channel-dependent scheduling** may be used to take advantage of channel state information. If the channel conditions are favourable, the throughput and system spectral efficiency may be increased. In even more advanced systems such as LTE, the scheduling is combined by channel-dependent packet-by-packet dynamic channel allocation, or by assigning OFDMA multi-carriers or other frequency-domain equalization components to the users that best can utilize them.

First in first out

Main article: First In First Out

Also known as ***F**irst **C**ome, **F**irst **S**erved* (FCFS), is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue.

- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, since long processes can hold the CPU
- Turnaround time, waiting time and response time can be high for the same reasons above
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization means that as long as every process eventually completes, there is no starvation. In an environment where some processes might not complete, there can be starvation.
- It is based on Queuing
- Here is the C-code for FCFS

Shortest remaining time

Main article: Shortest remaining time

Similar to ***S**hortest **J**ob **F**irst* (SJF). With this strategy the scheduler arranges processes with the least estimated processing time remaining to be next in the queue. This requires advanced knowledge or estimations about the time required for a process to complete.

- If a shorter process arrives during another process' execution, the currently running process may be interrupted (known as preemption), dividing that process into two separate computing blocks. This creates excess overhead through additional context switching. The scheduler must also place each incoming process into a specific place in the queue, creating additional overhead.
- This algorithm is designed for maximum throughput in most scenarios.
- Waiting time and response time increase as the process's computational requirements increase. Since turnaround time is based on waiting time plus processing time, longer processes are significantly affected by this. Overall waiting time is smaller than FIFO, however since no process has to wait for the termination of the longest process.
- No particular attention is given to deadlines, the programmer can only attempt to make processes with deadlines as short as possible.

- Starvation is possible, especially in a busy system with many small processes being run.
- This policy is no more in use.
- To use this policy we should have at least two processes of different priority

Fixed priority pre-emptive scheduling

Main article: Fixed priority pre-emptive scheduling

The OS assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes.

- Overhead is not minimal, nor is it significant.
- FPPS has no particular advantage in terms of throughput over FIFO scheduling.
- If the number of rankings is limited it can be characterized as a collection of FIFO queues, one for each priority ranking. Processes in lower-priority queues are selected only when all of the higher-priority queues are empty.
- Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times.
- Deadlines can be met by giving processes with deadlines a higher priority.
- Starvation of lower priority processes is possible with large amounts of high priority processes queuing for CPU time.

Round-robin scheduling

Main article: Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them.

- RR scheduling involves extensive overhead, especially with a small time unit.
- Balanced throughput between FCFS and SJF, shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJF.
- Poor average response time, waiting time is dependent on number of processes, and not average process length.
- Because of high waiting times, deadlines are rarely met in a pure RR system.
- Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

Multilevel queue scheduling

Main article: Multilevel feedback queue

This is used for situations in which processes are easily divided into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time

requirements and so may have different scheduling needs. It is very useful for shared memory problems.

Scheduling optimization problems

- Open-shop scheduling
- Job Shop Scheduling
- Flow Shop Scheduling Problem

Manual scheduling

Main article: Manual scheduling

A very common method in embedded systems is to manually schedule jobs. This can for example be done in a time-multiplexed fashion. Sometimes the kernel is divided in three or more parts: Manual scheduling, preemptive and interrupt level. Exact methods for scheduling jobs are often proprietary.

- No resource starvation problems.
- Very high predictability; allows implementation of hard real-time systems.
- Almost no overhead.
- May not be optimal for all applications.
- Effectiveness is completely dependent on the implementation.

How to choose a scheduling algorithm

When designing an operating system, a programmer must consider which scheduling algorithm will perform best for the use the system is going to see. There is no universal “best” scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above. For example, Windows NT/XP/Vista uses a multilevel feedback queue, a combination of fixed priority preemptive scheduling, round-robin, and first in first out. In this system, threads can dynamically increase or decrease in priority depending on if it has been serviced already, or if it has been waiting extensively. Every priority level is represented by its own queue, with round-robin scheduling amongst the high priority threads and FIFO among the lower ones. In this sense, response time is short for most threads, and short but critical system threads get completed very quickly. Since threads can only use one time unit of the round robin in the highest priority queue, starvation can be a problem for longer high priority threads.

Operating system process scheduler implementations

The algorithm used may be as simple as round-robin in which each process is given equal time (for instance 1 ms, usually between 1 ms and 100 ms) in a cycling list. So, process A executes for 1 ms, then process B, then process C, then back to process A.

More advanced algorithms take into account process priority, or the importance of the process. This allows some processes to use more time than other processes. The kernel always uses

whatever resources it needs to ensure proper functioning of the system, and so can be said to have infinite priority. In SMP(symmetric multiprocessing) systems, processor affinity is considered to increase overall system performance, even if it may cause a process itself to run more slowly. This generally improves performance by reducing cache thrashing.

Windows

Very early MS-DOS and Microsoft Windows systems were non-multitasking, and as such did not feature a scheduler. Windows 3.1x used a non-preemptive scheduler, meaning that it did not interrupt programs. It relied on the program to end or tell the OS that it didn't need the processor so that it could move on to another process. This is usually called cooperative multitasking. Windows 95 introduced a rudimentary preemptive scheduler; however, for legacy support opted to let 16 bit applications run without preemption.[1]

Windows NT-based operating systems use a multilevel feedback queue. 32 priority levels are defined, 0 through to 31, with priorities 0 through 15 being "normal" priorities and priorities 16 through 31 being soft real-time priorities, requiring privileges to assign. 0 is reserved for the Operating System. Users can select 5 of these priorities to assign to a running application from the Task Manager application, or through thread management APIs. The kernel may change the priority level of a thread depending on its I/O and CPU usage and whether it is interactive (i.e. accepts and responds to input from humans), raising the priority of interactive and I/O bounded processes and lowering that of CPU bound processes, to increase the responsiveness of interactive applications.[2] The scheduler was modified in Windows Vista to use the cycle counter register of modern processors to keep track of exactly how many CPU cycles a thread has executed, rather than just using an interval-timer interrupt routine.[3] Vista also uses a priority scheduler for the I/O queue so that disk defragmenters and other such programs don't interfere with foreground operations.[4]

Mac OS

Mac OS 9 uses cooperative scheduling for threads, where one process controls multiple cooperative threads, and also provides preemptive scheduling for MP tasks. The kernel schedules MP tasks using a preemptive scheduling algorithm. All Process Manager processes run within a special MP task, called the "blue task". Those processes are scheduled cooperatively, using a round-robin scheduling algorithm; a process yields control of the processor to another process by explicitly calling a blocking function such as WaitNextEvent. Each process has its own copy of the Thread Manager that schedules that process's threads cooperatively; a thread yields control of the processor to another thread by calling YieldToAnyThread or YieldToThread.[5]

Mac OS X uses a multilevel feedback queue, with four priority bands for threads - normal, system high priority, kernel mode only, and real-time.[6] Threads are scheduled preemptively; Mac OS X also supports cooperatively scheduled threads in its implementation of the Thread Manager in Carbon.[5]

AIX

In AIX Version 4 there are three possible values for thread scheduling policy :

- First In First Out : Once a thread with this policy is scheduled, it runs to completion unless it is blocked, it voluntarily yields control of the CPU, or a higher-priority thread becomes dispatchable. Only fixed-priority threads can have a FIFO scheduling policy.
- Round Robin: This is similar to the AIX Version 3 scheduler round-robin scheme based on 10ms time slices. When a RR thread has control at the end of the time slice, it moves to the tail of the queue of dispatchable threads of its priority. Only fixed-priority threads can have a Round Robin scheduling policy.
- OTHER This policy is defined by POSIX1003.4a as implementation-defined. In AIX Version 4, this policy is defined to be equivalent to RR, except that it applies to threads with non-fixed priority. The recalculation of the running thread's priority value at each clock interrupt means that a thread may lose control because its priority value has risen above that of another dispatchable thread. This is the AIX Version 3 behavior.

Threads are primarily of interest for applications that currently consist of several asynchronous processes. These applications might impose a lighter load on the system if converted to a multithreaded structure.

AIX 5 implements the following scheduling policies: FIFO, round robin, and a fair round robin. The FIFO policy has three different implementations: FIFO, FIFO2, and FIFO3. The round robin policy is named SCHED_RR in AIX, and the fair round robin is called SCHED_OTHER. This link provides additional information on AIX 5 scheduling:

Linux

Linux 2.4

In Linux 2.4, an O(n) scheduler with a multilevel feedback queue with priority levels ranging from 0-140 was used. 0-99 are reserved for real-time tasks and 100-140 are considered nice task levels. For real-time tasks, the time quantum for switching processes was approximately 200 ms, and for nice tasks approximately 10 ms.[*citation needed*] The scheduler ran through the run queue of all ready processes, letting the highest priority processes go first and run through their time slices, after which they will be placed in an expired queue. When the active queue is empty the expired queue will become the active queue and vice versa.

However, some Enterprise Linux distributions such as SUSE Linux Enterprise Server replaced this scheduler with a backport of the O(1) scheduler (which was maintained by Alan Cox in his Linux 2.4-ac Kernel series) to the Linux 2.4 kernel used by the distribution.

Linux 2.6.0 to Linux 2.6.22

From versions 2.6 to 2.6.22, the kernel used an O(1) scheduler developed by Ingo Molnar and many other kernel developers during the Linux 2.5 development. For many kernel in time frame, Con Kolivas developed patch sets which improved interactivity with this scheduler or even replaced it with his own schedulers.

Since Linux 2.6.23

Con Kolivas's work, most significantly his implementation of "fair scheduling" named "Rotating Staircase Deadline", inspired Ingo Molnár to develop the Completely Fair Scheduler as a replacement for the earlier O(1) scheduler, crediting Kolivas in his announcement.[7]

The Completely Fair Scheduler (CFS) uses a well-studied, classic scheduling algorithm called fair queuing originally invented for packet networks. Fair queuing had been previously applied to CPU scheduling under the name stride scheduling.

The fair queuing CFS scheduler has a scheduling complexity of O(log N), where N is the number of tasks in the runqueue. Choosing a task can be done in constant time, but reinserting a task after it has run requires O(log N) operations, because the run queue is implemented as a red-black tree.

CFS is the first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.

The Brain Fuck Scheduler (BFS) is an alternative to the CFS.

FreeBSD

FreeBSD uses a multilevel feedback queue with priorities ranging from 0-255. 0-63 are reserved for interrupts, 64-127 for the top half of the kernel, 128-159 for real-time user threads, 160-223 for time-shared user threads, and 224-255 for idle user threads. Also, like Linux, it uses the active queue setup, but it also has an idle queue.[9]

NetBSD

NetBSD uses a multilevel feedback queue with priorities ranging from 0-223. 0-63 are reserved for time-shared threads (default, SCHED_OTHER policy), 64-95 for user threads which entered kernel space, 96-128 for kernel threads, 128-191 for user real-time threads (SCHED_FIFO and SCHED_RR policies), and 192-223 for software interrupts.

Solaris

Solaris uses a multilevel feedback queue with priorities ranging from 0-169. 0-59 are reserved for time-shared threads, 60-99 for system threads, 100-159 for real-time threads, and 160-169 for low priority interrupts. Unlike Linux, when a process is done using its time quantum, it's given a new priority and put back in the queue.

POSIX real time operating system.

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

INTER PROCESS COMMUNICATION MECHANISMS

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 ***nonblocking***. After sending a blocking communication, the process goes into the waiting state until it receives a response. Nonblocking communication allows the process to continue execution after sending the communication. Both types of communication are useful. There are two major styles of interprocess communication: ***shared memory*** and ***message passing***. The two are logically equivalent—given one, you can build an interface that implements the other. However, some programs may be easier to write using one rather than the other. In addition, the hardware platform may make one easier to implement or more efficient than the other.

Shared Memory Communication

Figure 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.

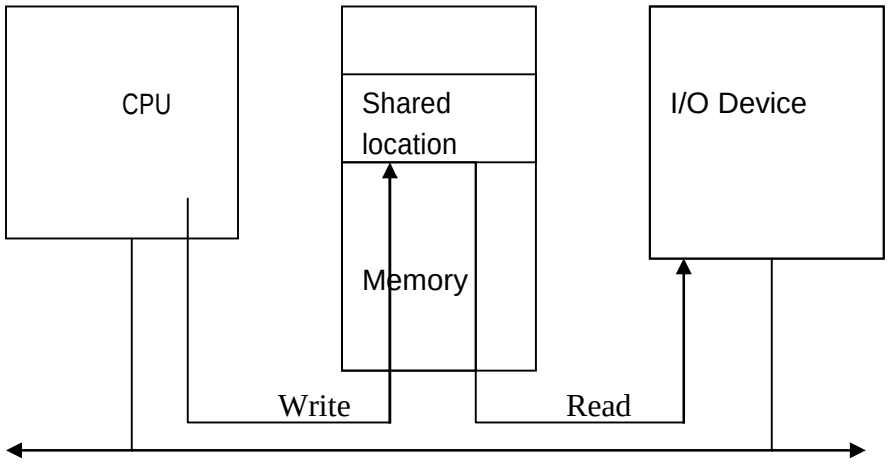


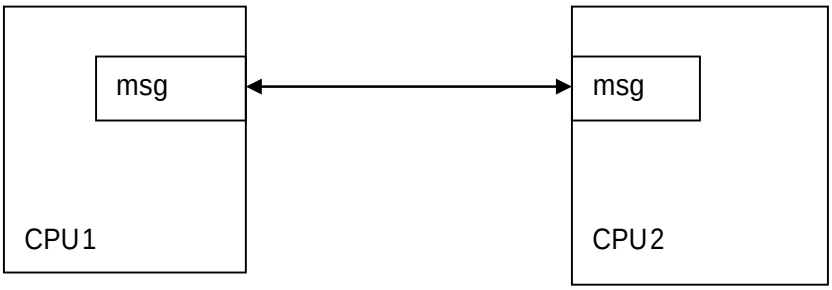
Fig : Shared memory communication implemented on a bus

As an application of shared memory, let us consider the situation of Figure in 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. The CPU, for example,would write the data, and then set the flag location to 1. 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.

The above scenario is caused by a critical timing race between the two programs. To avoid such problems, the microprocessor bus must support an atomic ***test-and set*** operation, which is available on a number of microprocessors. The test-and-set operation first reads a location and then sets it to a specified value. It returns the result of the test. If the location was already set, then the additional set has no effect but the test-and-set instruction returns a false result. If the location was not set, the instruction returns true and the location is in fact set. The bus supports this as an ***atomic*** operation that cannot be interrupted.

MESSAGE PASSING:



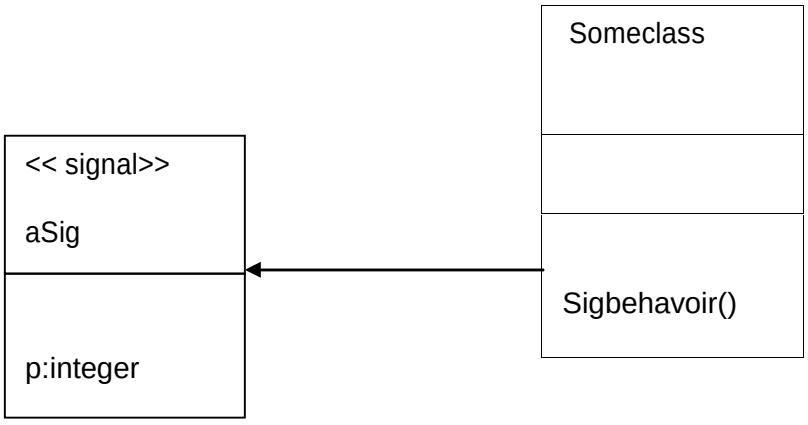
Message passing communication complements the shared memory model. As shown in Figure, each communicating entity has its own message send/receive unit. The message is not stored on

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. 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. 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 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.



USE OF A UML SIGNAL

EVALUATING OPERATING SYSTEM PERFORMANCE

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

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.

POWER OPTIMIZATION STRATEGY FOR PROCESSES

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** 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. Astraightforward 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.

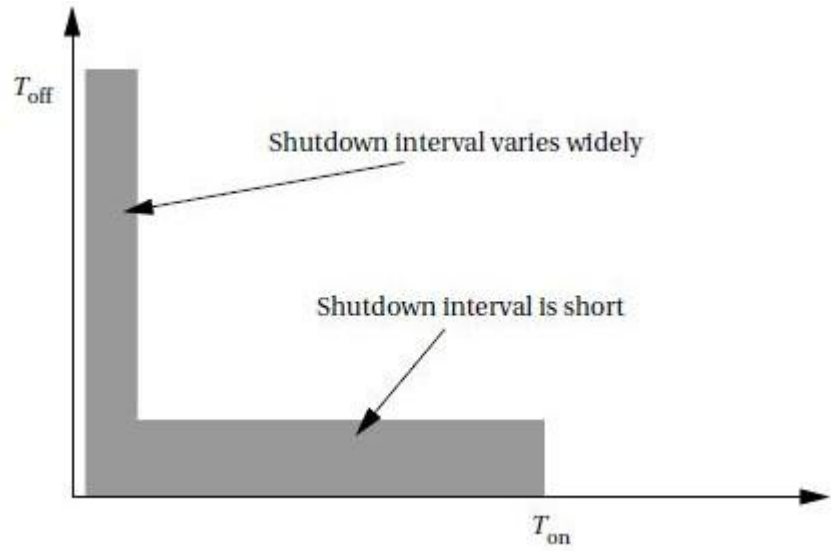


Fig An L-shaped usage distribution.

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 .

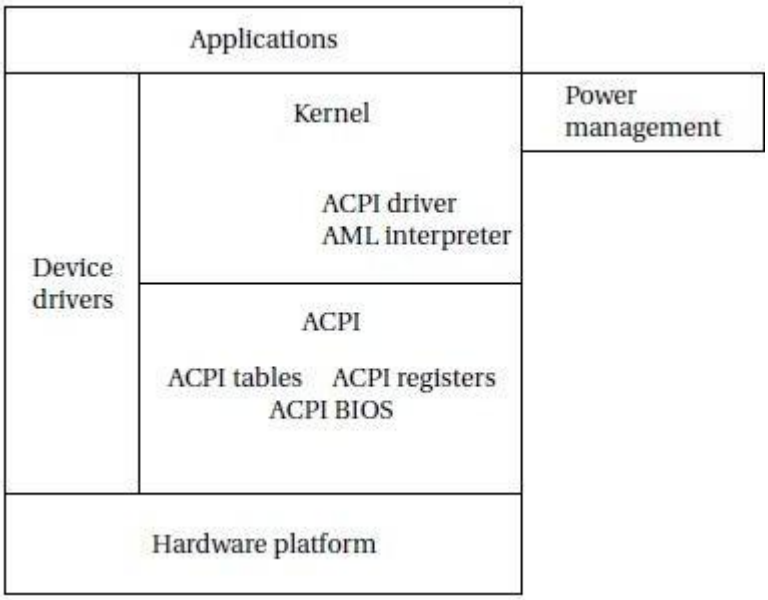


Fig The advanced configuration and power interface and its relationship to a complete system.

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.

IMPORTANT QUESTIONS
PART-A (2 MARKS)

1. **Define process.**
2. **Define thread.**
3. **Mention the requirements on processes.**
4. **Define period.**
5. **Define task graph.**
6. **Define initiation time and completion time.**
7. **Mention the various scheduling states of a process.**
8. **Define scheduling policy.**
9. **Define utilization.**
10. **Define time quantum.**
11. **Define context switching.**
12. **Mention the two ways of assigning priority to a process.**
13. **Define rate monolithic scheduling.**
14. **Define earliest deadline first scheduling.**
15. **Define priority inversion.**
16. **Mention the two different styles used for inter process communication.**
17. **Define signal.**
18. **Define response time.**
19. **Define PCB.**
20. **Define critical instant.**

PART– B(16 MARKS)

1. **Explain multiple tasks and multiple processes in detail.**
2. **Explain the various scheduling policies in detail.**
3. **Explain Preemptive real time operating system in detail.**
4. **Explain Non-Preemptive real time operating systems in detail.**
5. **Explain priority based scheduling in detail.**
6. **Explain the various inter process communication mechanism in detail.**
7. **Explain the various types of Performance issues.**

EMBEDDED SYSTEMS

SYSTEM DESIGN TECHNIQUES AND NETWORKS

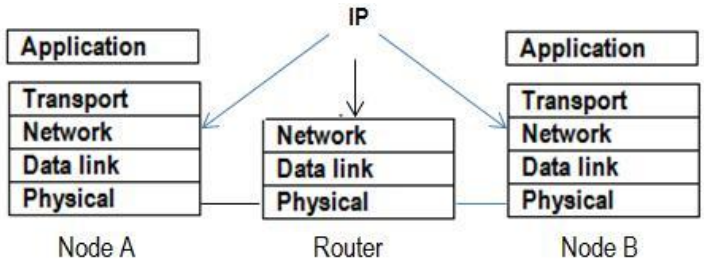
UNIT-4

What do you mean by quality and quality assurance related to embedded systems

The quality of a product or service can be judged by how well it satisfies its intended function. A product can be of low quality for several reasons, such as it was shoddily manufactured, its components were improperly designed, its architecture was poorly conceived, and the product's requirements were poorly understood.

Quality must be designed in. the bugs cannot be tested enough to deliver a high-quality product. The quality assurance (QA) process is vital for the delivery of a satisfactory system.

Give examples of internet enabled system.



List the OSI layers from lowest to highest level of abstraction.

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.

What is a distributed embedded architecture

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

What are the merits of embedded distributed architecture?

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

6. Differentiate counter semaphore and binary semaphores.

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

What is priority inheritance?

Priority inheritance is a method of eliminating priority inversion, using this a 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.

Briefly discuss about the design methodologies for an embedded computing system.

The design methodologies for an embedded computing system:

The goal of a design process is to create a product that does something 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 products is 3–6 months—if you are 3 months late, you will never make money.

- 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.

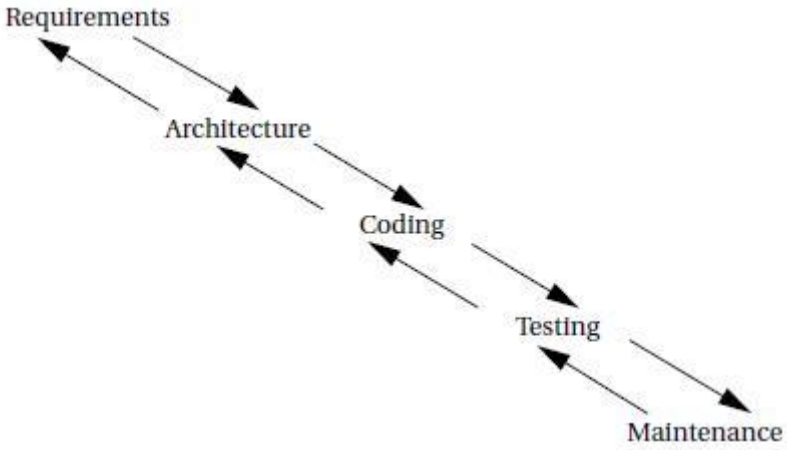
- Quality: Customers not only want their products fast and cheap, they also want them to be right. A design methodology that cranks out shoddy products will soon be forced out of the marketplace. Correctness, reliability, and usability must be explicitly addressed from the beginning of the design job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

Design Flows:

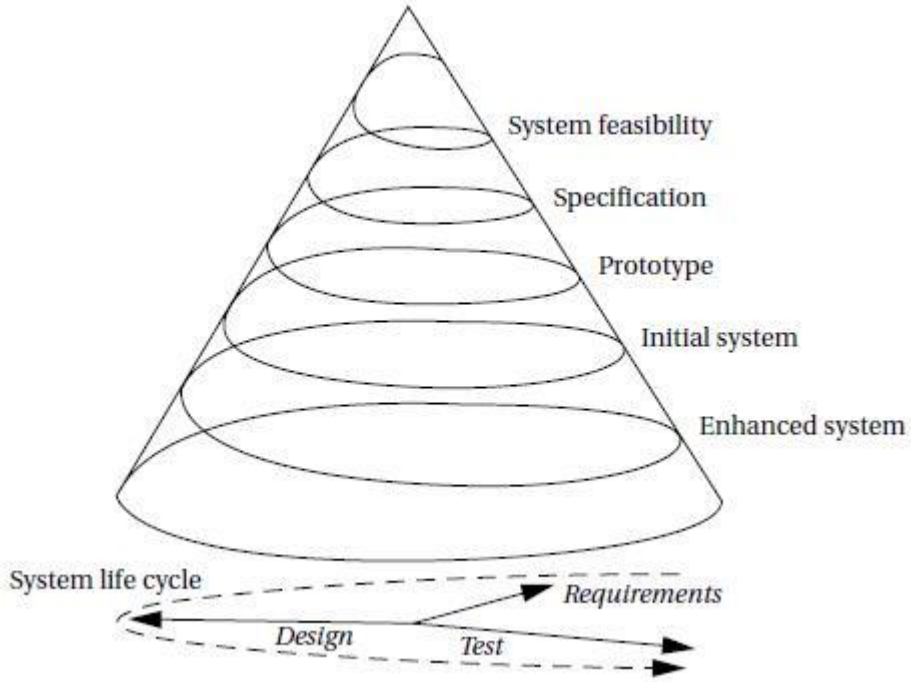
A **design flow** is a sequence of steps to be followed during a design. Some of the steps can be performed by tools, such as compilers or CAD systems; other steps can be performed by hand. In this section we look at the basic characteristics of design flows.

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 functionality 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.



The waterfall model of software development.



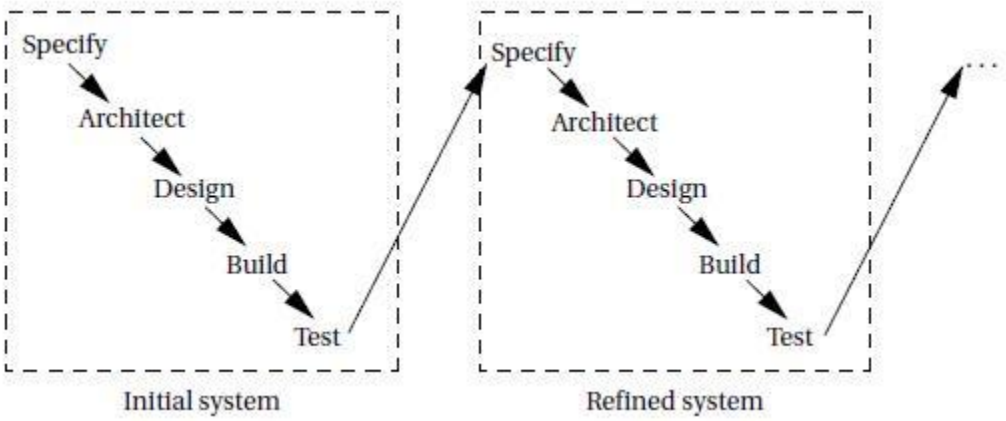
The spiral model of software design.

The spiral model is an alternative model of software development. 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 spiral 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.

Successive Refinement Design:

A **successive refinement** design methodology is an approach in which the system is built several times. A first system is used as a rough prototype, 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

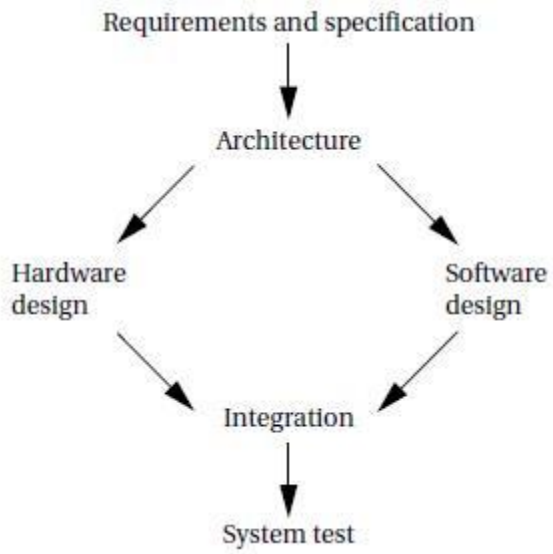
are building the system. Refining the system by building several increasingly complex systems allows you to test out architecture and design techniques. The various iterations may also be only partially completed; for example, continuing an initial system only through the detailed design phase may teach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



A successive refinement development model.

A simple hardware/software design methodology:

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.

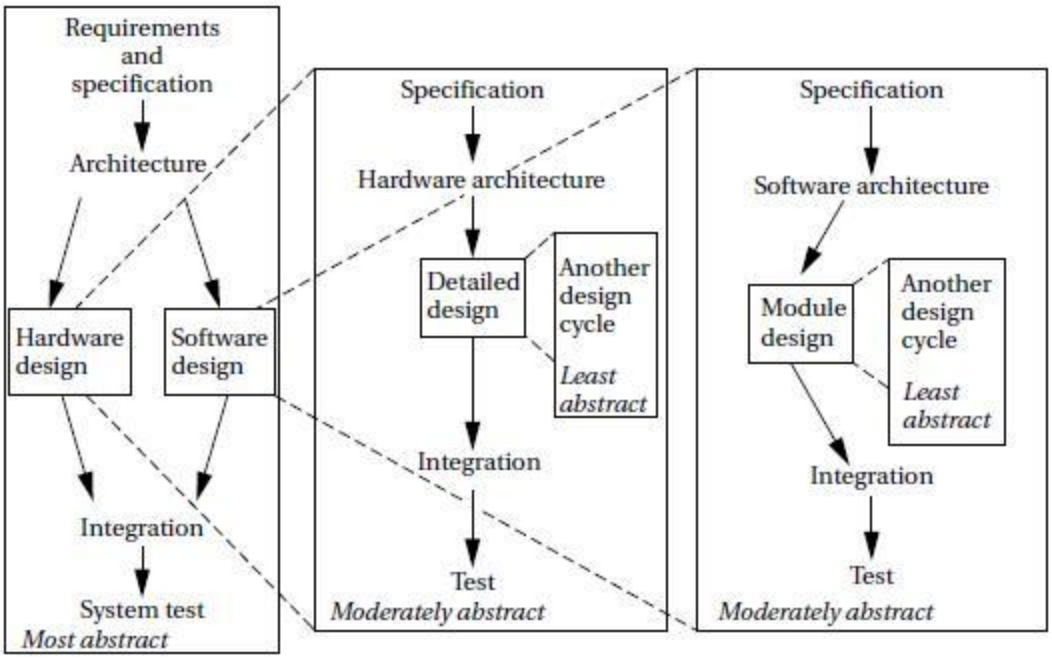


A simple hardware/software design methodology.

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. Reduceddesign 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 between design and manufacturing.



A hierarchical design flow for an embedded system.

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.

Discuss in detail about the network based embedded system.

NETWORKS FOR EMBEDDED SYSTEMS:

Networks for embedded computing span a broad range of requirements; many of those requirements are very different from those for general-purpose networks. Some networks are used in safety-critical applications, such as automotive control. Some networks, such as those used in consumer electronics systems, must be very inexpensive. Other networks, such as industrial control networks, must be extremely rugged and reliable.

Several interconnect networks have been developed especially for distributed embedded computing:

■ The I^2C bus is used in microcontroller-based systems.

■ The Controller Area Network (CAN) bus was developed for automotive electronics. It provides megabit rates and can handle large numbers of devices.

■ Ethernet and variations of standard Ethernet are used for a variety of control appliations.

The I^2C Bus:

- The **I^2C bus** [Phi92] is a well-known bus commonly used to link microcontrollers into systems. It has even been used for the command interface in an MPEG-2 video chip [van97]; while a separate bus was used for high-speed video data, setup information was transmitted to the on-chip controller through an I^2C bus interface.

- I^2C is designed to be low cost, easy to implement, and of moderate speed (up to 100 KB/s for the standard bus and up to 400 KB/s for the extended bus).

- As a result, it uses only two lines: the **serial data line (SDL)** for data and the **serial clockline (SCL)**, which indicates when valid data are on the data line. Figure 4.6 shows thestructure of a typical I^2C bus system.

- Every node in the network is connected to both SCL and SDL. Some nodes may be able to act as bus masters and the bus may have more than one master. Other nodes may act as slaves that only respond to requests from masters.

- The basic electrical interface to the bus is shown in Figure 4.7.The bus does not define particular voltages to be used for high or low so that either bipolar or MOS circuits can be connected to the bus.

- Both bus signals use open collector/open drain circuits.1 A pull-up resistor keeps the default state of the signal high, and transistors are used in each bus device to pull down the signal when a 0 is to be transmitted.

- The Open collector/open drain signaling allows several devices to simultaneously write the bus without causing electrical damage. The open collector/open drain circuitry allows a slave device to stretch a clock signal during a read from a slave. The master is responsible for generating the SCL clock, but the slave can stretch the low period of the clock (but not the high period) if necessary.

- The I2C interface on a microcontroller can be implemented with varying percentages of the functionality in software and hardware [Phi89]. As illustrated in Figure, a typical system has a 1-bit hardware interface with routines for byte level functions.

- The I_2C device takes care of generating the clock and data. The application code calls routines to send an address, send a data byte, and so on, which then generates the SCL and SDL, acknowledges, and so forth.

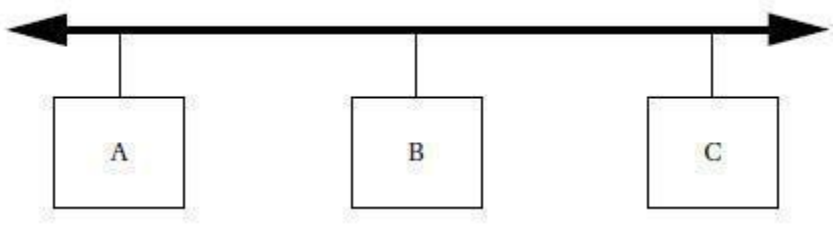
- One of the microcontroller's timers is typically used to control the length of bits on the bus.
- Interrupts may be used to recognize bits. However, when used in master mode, polled I/O may be acceptable if no other pending tasks can be performed, since masters initiate their own transfers.

Ethernet

- Ethernet is very widely used as a local area network for general-purpose computing. Because of its ubiquity and the low cost of Ethernet interfaces, it has seen significant use as a network for embedded computing.

- Ethernet is particularly useful when PCs are used as platforms, making it possible to use standard components, and when the network does not have to meet rigorous real-time requirements.

- The physical organization of an Ethernet is very simple, as shown in figure. The network is a bus with a single signal path; the Ethernet standard allows for several different implementations such as twisted pair and coaxial cable.



Ethernet organization.

- Unlike the I_2C bus,nodes on the Ethernet are not synchronized they can send their bits at any time. I_2C relies on the fact that a collision can be detected and quashed within a single bit time thanks to synchronization.

- But since Ethernet nodes are not synchronized, if two nodes decide to transmit at the same time, the message will be ruined. The Ethernet arbitration scheme is known as **Carrier Sense Multiple Access** with **Collision Detection (CSMA/CD)**.

Field bus

- Manufacturing systems require networked sensors and actuators. Field bus (http://www.fieldbus.org) is a set of standards for industrial control and instrumentation systems.

- The H1 standard uses a twisted-pair physical layer that runs at 31.25 MB/s. It is designed for device integration and process control. The High Speed Ethernet standard is used for backbone networks in industrial plants. It is based on the 100 MB/s Ethernet standard. It can integrate devices and subsystems.

NETWORK-BASED DESIGN:

- Designing a distributed embedded system around a network inwvwowl.vanenasunsivoemrzitey.com the same design tasks we faced in accelerated systems. We must schedule computations in time and allocate them to PEs. Scheduling and allocation of communication are important additional design tasks required for many distributed networks.

- Many embedded networks are designed for low cost and therefore do not provide excessively high communication speed. If we are not careful, the network can become the bottleneck in system design. In this section we concentrate on design tasks unique to network-based distributed embedded systems.

- We know how to analyze the execution time of programs and systems of processes on single CPUs, but to analyze the performance of networks we must know how to determine the delay incurred by transmitting messages. Let us assume for the moment that messages are sent reliably we do not have to retransmit a message.

- The **message delay** for a single message with no contention (as would be the case in a point-to-point connection) can be modeled as

$$t_m = t_x + t_n + t_r$$

- where t_x is the transmitter-side overhead, t_n is the network transmission time, and t_r is the receiver-side overhead. In I^2C, t_x and t_r are negligible relative to t_n

- If messages can interfere with each other in the network, analyzing communication delay becomes difficult. In general, because we must wait for the network to become available and then transmit the message, we can write the **message delay** as

$$t_y = t_d + t_m$$

where td is the **network availability delay** incurred waiting for the network to become available. The main problem, therefore, is calculating t_d. That value depends on the type of arbitration used in the network.

- If the network uses fixed-priority arbitration, the network availability delay is unbounded for all but the highest-priority device. Since the highest-priority device always gets the network first, unless there is an application-specific limit on how long it will transmit before relinquishing the network, it can keep blocking the other devices indefinitely.

- If the network uses fair arbitration, the network availability delay is bounded. In the case of round-robin arbitration, if there are N devices, then the worst case network availability delay is $N(t_x+t_{arb})$,where t_{arb} is the delay incurred for arbitration. t_{arb} is usually small compared to transmission time.

- Of course,a round-robin arbitrated network puts all communications at the same priority. This does not eliminate the priority inversion problem because processes still have priorities. Thus far we have assumed a **single-hop network**: A message is received at its intended destination directly from the source,without going through any other network node.

- It is possible to build **multihop networks** in which messages are routed through network nodesto get to their destinations. (Using a multistage network does not necessarily mean using a multihop network—the stages in a multistage network are generally much smaller than the network PEs.) Figure shows an example of a multihop communication.

- The hardware platform has two separate networks (perhaps so that communications between subsets of the PEs do not interfere),but there is no direct path from M1 to M5.The message is therefore routed through M3, which reads it from one network and sends it on to the other one.

- Analyzing delays through multihop systems is very difficult. For example,the time that the message is held at M3 depends on both the computational load of M3 and the other messages that it must handle.

- If there is more than one network,we must allocate communications to the networks. We may establish multiple networks so that lower-priority communications can be handled separately without interfering with high-priority communications on the primary network.

- Scheduling and allocation of computations and communications are clearly interrelated. If we change the allocation of computations, we change not only the scheduling of processes on those PEs but also potentially the schedules of PEs with which they communicate.

- For example, if we move a computation to a slower PE, its results will be available later, which may mean rescheduling both the process that uses the value and the communication that sends the value to its destination.

Write notes on internet enabled systems.

Explain networks for embedded systems and Internet-enabled embedded system.

Explain how Internet can be used by embedded computing systems.

Discuss about Internet enabled systems and architecture of distributed embedded systems.

INTERNET-ENABLED SYSTEMS:

Some very different types of distributed embedded system are rapidly emerging the

Internet- enabled embedded system and **Internet appliances**. The Internet is notwell suited to the real- time tasks that are the bread and butter of embedded computing, but it does provide a rich environment for non–real-time interaction. In this section we will discuss the Internet and how it can be used by embedded computing systems.

Internet

- The **Internet Protocol (IP)** [Los97, Sta97A] is the fundamental protocol on the **Internet**. It provides connectionless, packet-based communication. Industrial automation has long been a good application area for Internet-based embedded systems.

- Information appliances that use the Internet are rapidly becoming another use of IP in embedded computing. Internet protocol is not defined over a particular physical implementation it is an **internetworking** standard. Internet packets are assumed to be carried by some other network, such as an Ethernet. In general, an Internet packet will travel over several different networks from source to destination.

- The IP allows data to flow seamlessly through these networks from one end user to another. The relationship between IP and individual networks is illustrated in Figure 4.6. IP works at the network layer.

- When node A wants to send data to node B, the application's data pass through several layers of the protocol stack to send to the IP. IP creates packets for routing to the destination, which are then sent to the data link and physical layers. A node that transmits data among different types of networks is known as a **router**.

- The router's functionality must go up to the IP layer, but since it is not running applications, it does not need to go to higher levels of the OSI model.

- In general, a packet may go through several routers to get to its destination. At the destination, the IP layer provides data to the transport layer and ultimately the receiving application.

- As the data pass through several layers of the protocol stack, the IP packet data are encapsulated in packet formats appropriate to each layer.

- The basic format of an IP packet is shown in figure. The header and data payload are both of variable length. The maximum total length of the header and data payload is 65,535 bytes.

- An Internet address is a number (32 bits in early versions of IP, 128 bits in IPv6). The IP address is typically written in the form xxx.xx.xx.xx. The names by which users and applications typically refer to Internet nodes, such as foo.baz.com, are translated into IP addresses via calls to a **Domain Name Server**, one of the higher-level services built on top of IP.

- The fact that IP works at the network layer tells us that it does not guarantee that a packet is delivered to its destination. Furthermore, packets that do arrive may come out of order. This is referred to as **best-effort routing**.

- Since routes for data may change quickly with subsequent along very different paths with different delays, real-time performance of IP can be hard to predict.

- The **Transmission Control Protocol (TCP)** is one such example. It provides a connection oriented service that ensures that data arrive in the appropriate order, and it uses an acknowledgment protocol to ensure that packets arrive. Because many higher - level services are built on top of TCP, the basic protocol is often referred to as TCP/IP.

- The figure shows the relationships between IP and higher-level Internet services. Using IP as the foundation, TCP is used to provide **File Transport Protocol** for batch file transfers, **Hypertext Transport Protocol (HTTP)** for Worldwide Web service, **Simple Mail Transfer Protocol** for email, and Telnet for virtual terminals.

- A separate transport protocol, **User Datagram Protocol**, is used as the basis for the network management provided by the **Simple Network Management Protocol**

Internet Applications

- The Internet provides a standard way for an embedded system to act in concert with other devices and with users, such as:

- One of the earliest Internet-enabled embedded systems was the laser printer. High-end laser printers often use IP to receive print jobs from host machines.

- Portable Internet devices can display Web pages, read email, and synchronize calendar information with remote computers.

- A home control system allows the homeowner to remotely monitor and control home cameras, lights, and so on.

- Although there are higher-level services that provide more time-sensitive delivery mechanisms for the Internet, the basic incarnation of the Internet is not well suited on hard real-time operations. However, IP is a very good way to let the embedded system talk to other systems.

- IP provides a way for both special-purpose and standard programs (such as Web browsers) to talk to the embedded system. This non–real-time interaction can be used to monitor the system, set its configuration, and interact with it.

7. Discuss in detail about the distributed embedded architecture. (16) (N/D – 14)

8. Discuss in detail about the distributed embedded architecture with neat sketch. (16) (N/D – 13) 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. 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.

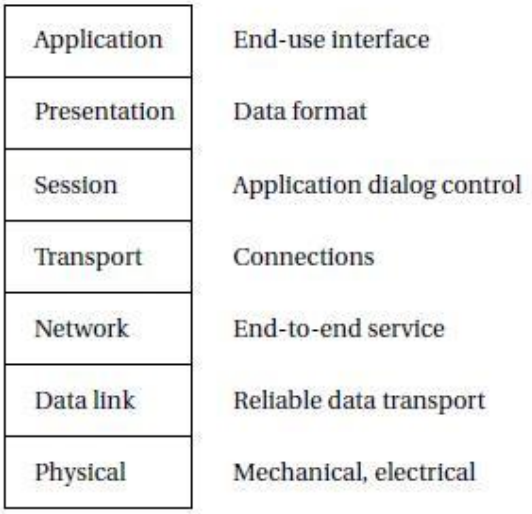
The system of PEs and networks forms the hardware platform on which the application runs. However, unlike the system bus of Chapter 4, the distributed embedded system does not have memory on the bus (unless a memory unit is organized as an I/O device that speaks the network protocol). In particular, 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.

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. The seven layers of the **OSI model**, shown in figure, 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 model layers.

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.

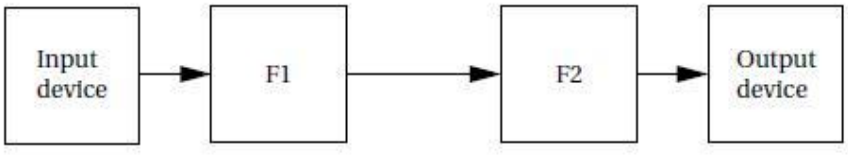
- Transport: The transport layer defines connection-oriented services that ensure that data are delivered in the proper order and without errors across multiple links. This layer may also try to optimize network resource utilization.
- Session: A session provides mechanisms for controlling the interaction 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 betweethe 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.

Hardware and Software Architectures:

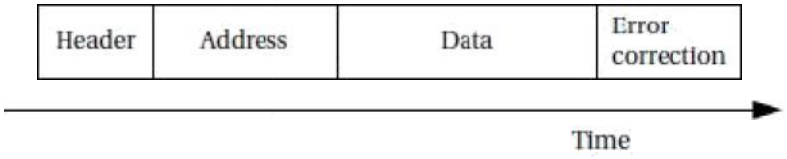
Distributed embedded systems can be organized in many different ways depending upon the needs of the application and cost constraints. One good way to understand possible architectures is to consider the different types of interconnection networks that can be used.

A **point-to-point** link establishes a connection between exactly two PEs. Point to-point links are simple to design precisely because they deal with only two components. We do not have to worry about other PEs interfering with communication on the link. The figure below shows a simple example of a distributed embedded system built from point-to-point links. The input signal is sampled by the input device and passed to the first digital filter, F1, over a point-to-point link. The results of that filter are sent through a second point-to-point link to filter F2. The results in turn are sent to the output device over a third point-to-point link.A digital filtering system requires that its outputs arrive at strict intervals, which means that the filters must process their inputs in a timely fashion. Using point-to-point connections allows both F1 and F2 to receive a new sample and send a new output at the same time without worrying about collisions on the communications network.



A signal processing system built from print-to-point links.

It is possible to build a **full-duplex**, point-to-point connection that can be used for simultaneous communication in both directions between the two PEs. (A halfduplex connection allows for only one-way communication.) A bus is a more general form of network since it allows multiple devices to be connected to it. Like a microprocessor bus, PEs connected to the bus have addresses. Communications on the bus generally take the form of **packets** as illustrated in Figure.



Format of a typical message on a bus.

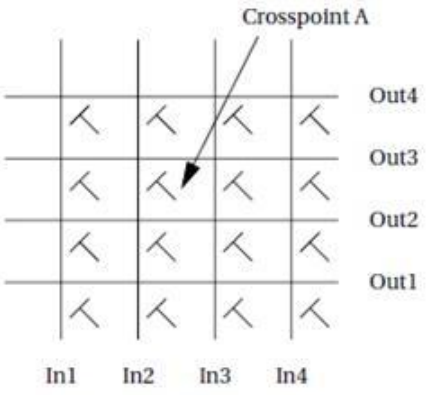
A packet contains an address for the destination and the data to be delivered. It frequently includes error detection/correction information such as parity. It also may include bits that serve to signal to other PEs that the bus is in use, such as the header shown in the figure. The data to be transmitted from one PE to another may not fit exactly into the size of the data payload on the packet. It is the responsibility of the transmitting PE to divide its data into packets; the receiving PE must of course reassemble the complete data message from the packets.

Distributed system buses must be arbitrated to control simultaneous access, just as with microprocessor buses. Arbitration scheme types are summarized below.

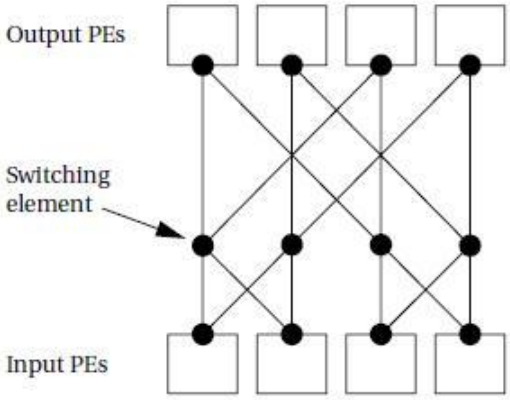
- Fixed-priority arbitration always gives priority to competing devices in the same way. If a high-priority and a low-priority device both have long data transmissions ready at the same time, it is quite possible that the low-priority device will not be able to transmit anything until the high-priority device has sent all its data packets.
- Fair arbitration schemes make sure that no device is starved. Round-robin arbitration is the most commonly used of the fair arbitration schemes. The PCI bus requires that the arbitration scheme used on the bus must be fair, although it does not specify a particular arbitration scheme. Most implementations of PCI use round-robin arbitration.

A bus has limited available bandwidth. Since all devices connect to the bus, communications can interfere with each other. Other network topologies can be used to reduce communication conflicts. At the opposite end of the generality spectrum from the bus is the crossbar network shown in Figure crossbar not only allows any input to be connected to any output, it also allows all combinations of input/output connections to be made. Many other networks have been designed that provide varying amounts of parallel communication at

varyinghardware costs. The second figure shows an example multistage network.



A crossbar network.



A multistage network.

EMBEDDED SYSTEMS
UNIT V- CASE STUDY

5.1 DATA COMPRESSOR:

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.

Requirements and Algorithm

We use the ***Huffman coding*** technique. We require some understanding of how our compression code fits into a larger system. Figure 5.1 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.Theoutput symbols are variable length, so we 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.

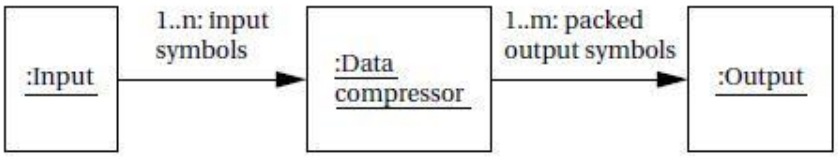


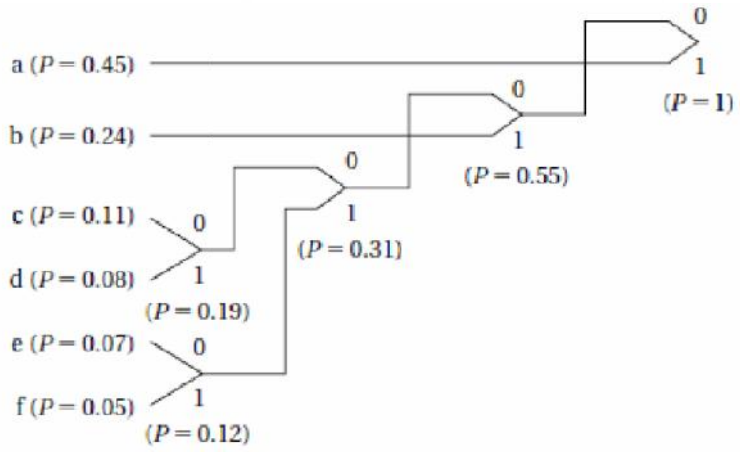
Fig 5.1 UML collaboration diagram for the data compressor

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 which makes use of informationon 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.

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 below.we obtain the following coding of the characters:

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

Requirements:

Name	Data compression module
Purpose	Code module for Huffman data compression
Inputs	Encoding table, uncodedbyte-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 5.1 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.

Class Diagram:

A class description for this refined understanding of the requirements on the module is shown in Figure 5.2. The class's *buffer* and *current-bit* behavior 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 returnstwo 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 currentcontents of the internal buffer.
■ ***Flush*** returns the current state of the buffer, including the number of valid bits in thebuffer.

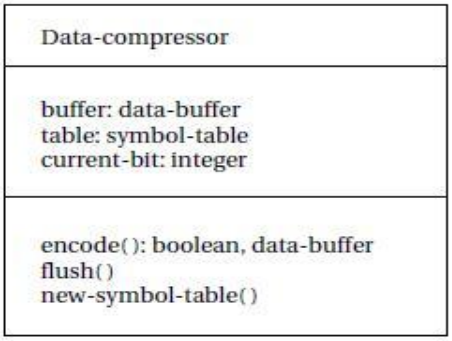


Fig 5.2 Definition of the data compressor class

We also need to define classes for the data buffer and the symbol table. These classes are shown in Figure 5.3.The *data-buffer* will be used to hold both packed symbols and unpacked ones (such as in the symbol table).

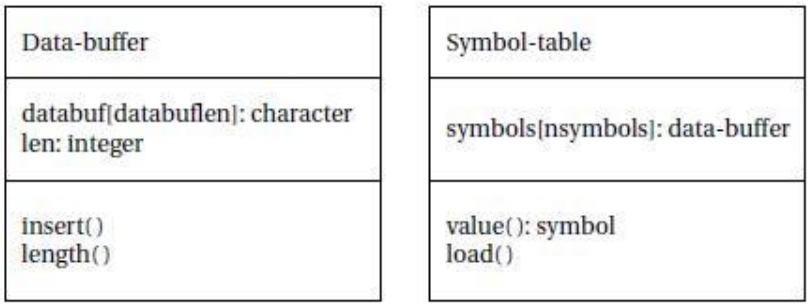


Fig 5.3 Additional class definition for the data compressor

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 indexesthe encoded version of each symbol.

The class defines an access behavior for the table; it also defines a *load* behavior to create a new symbol table. The relationships between these classes are shown in Figure 5.4—a data compressor object includes one buffer and one symbol table.

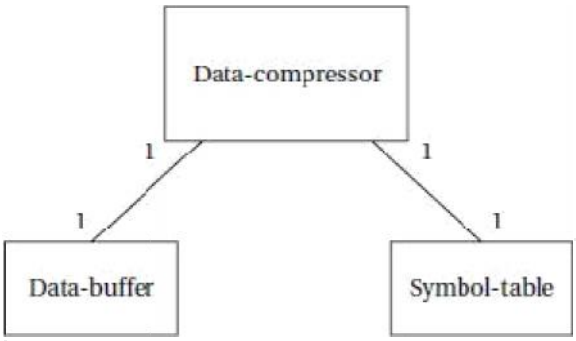


Fig 5.4 Relationship between classes in the data compressor

Figure 5.5 shows a state diagram for the *encode* behavior. It shows that most of the effort goes into filling the buffers with variable-length symbols.

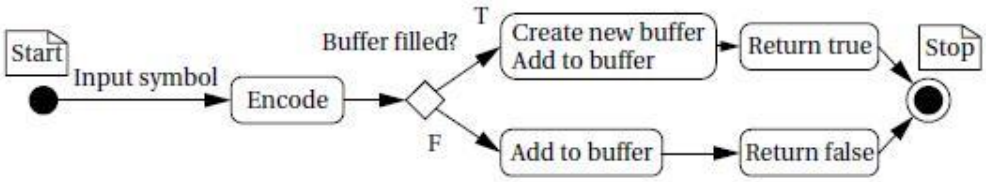


Fig 5.5 State diagram for encode behavior

Figure 5.6shows a state diagram for *insert*.

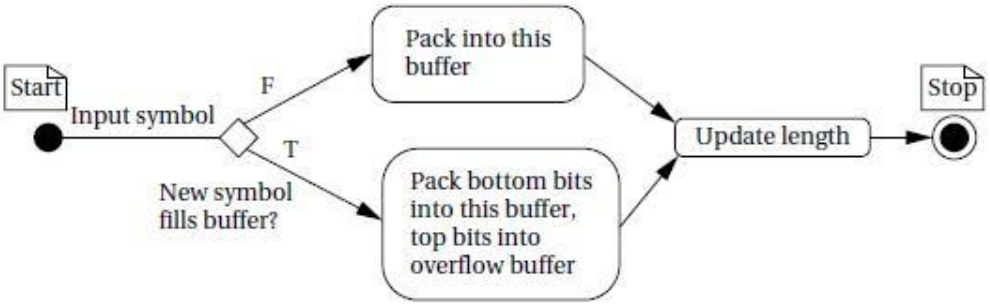


Fig 5.6 State diagram for insert behavior

5.2 ALARM CLOCK

An alarm clock, 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.

The basic functions of an alarm clock are well understood and easy to enumerate. Figure 5.7 illustrates the front panel design for the alarm clock.

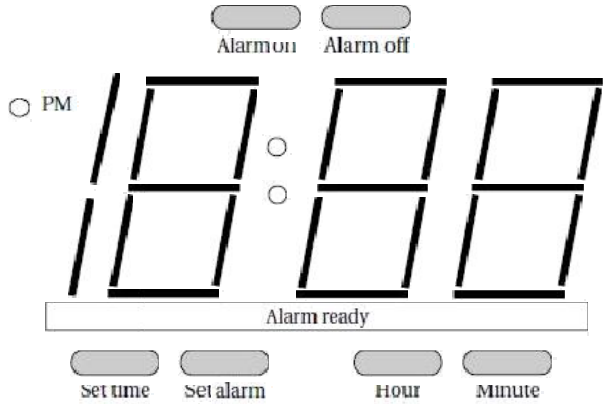


Fig 5.7 Front panel of the Alarm clock

Requirements

Name	**Alarm clock.**
Purpose	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.
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:

Figure 5.8 shows the basic classes for the alarm clock.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.

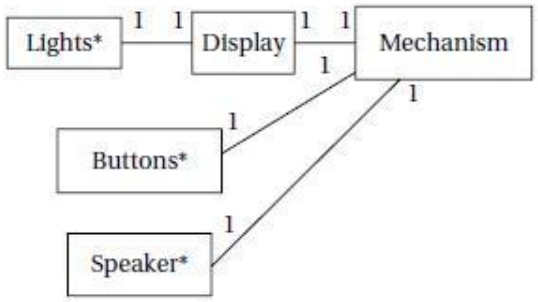


Fig 5.8 Class diagram for the Alarm clock

The details of the low-level user interface classes are shown in Figure 5.9

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.

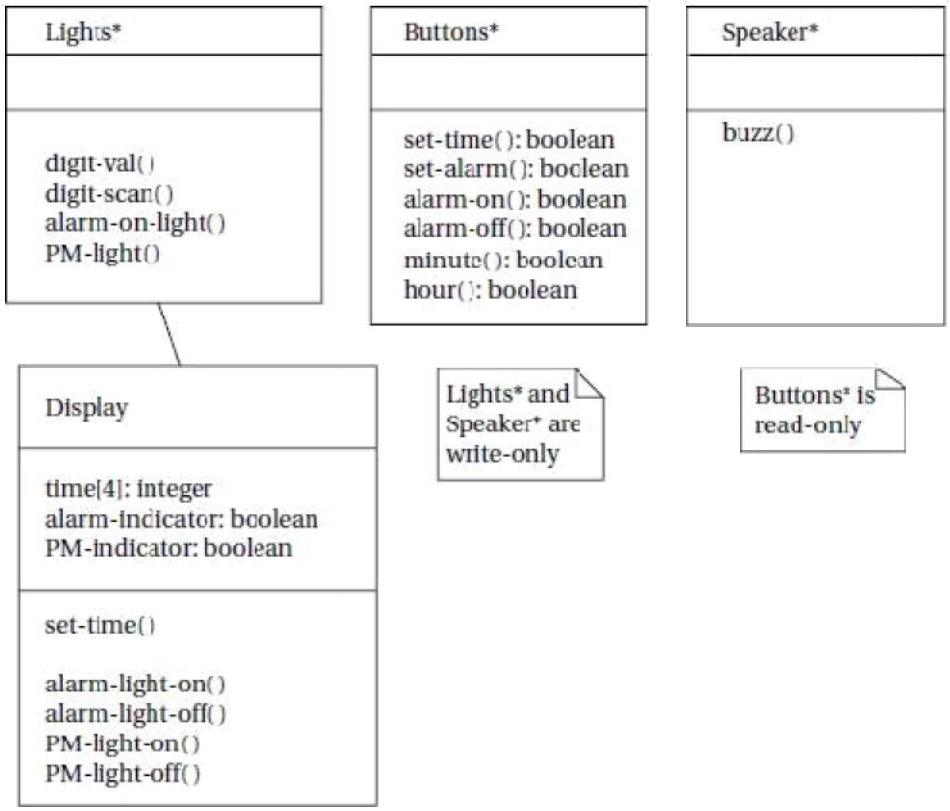


Fig 5.9 Details of Low level Class for the Alarm clock

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 5.10. 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.

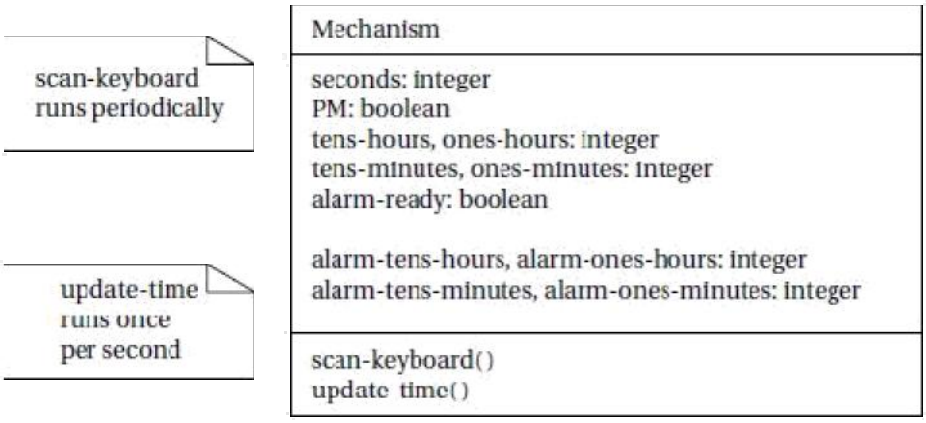


Fig 5.10 The mechanism class

Figure 5.11 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.

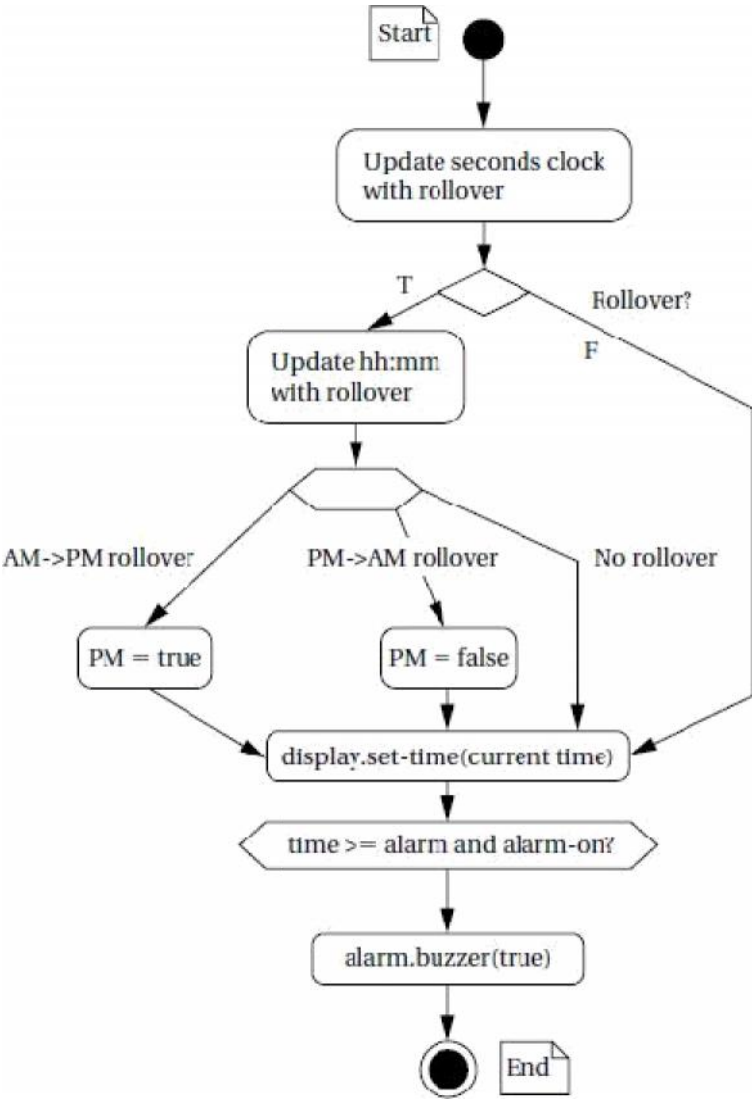


Fig 5.11 State diagram for update time

The state diagram for *scan-keyboard* is shown in Figure 5.12.

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.

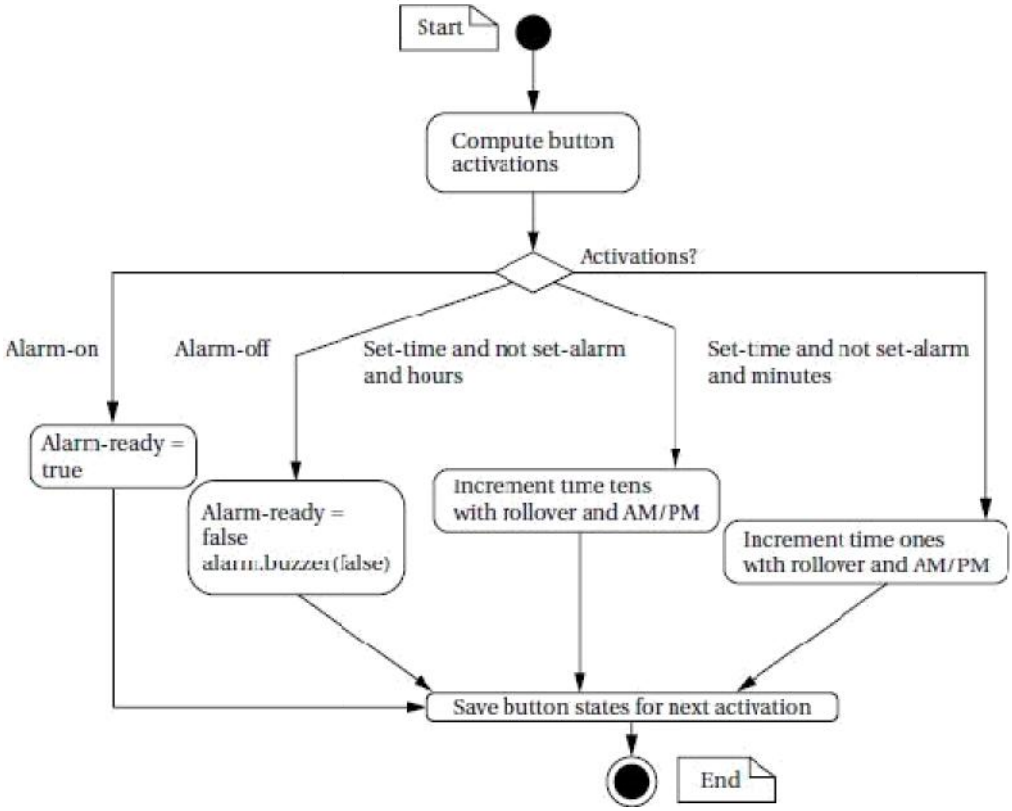


Fig 5.12 State diagram for scan keyboard

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 behavior 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. 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 avariable 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. Sincebuttons 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.

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 testbench for this code that generates button depressions to exercise the state machine.

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.

5.3 AUDIO PLAYERS

Audio players are often called ***MP3 players*** after the popular audio data format. The earliest portable MP3 playerswere 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.

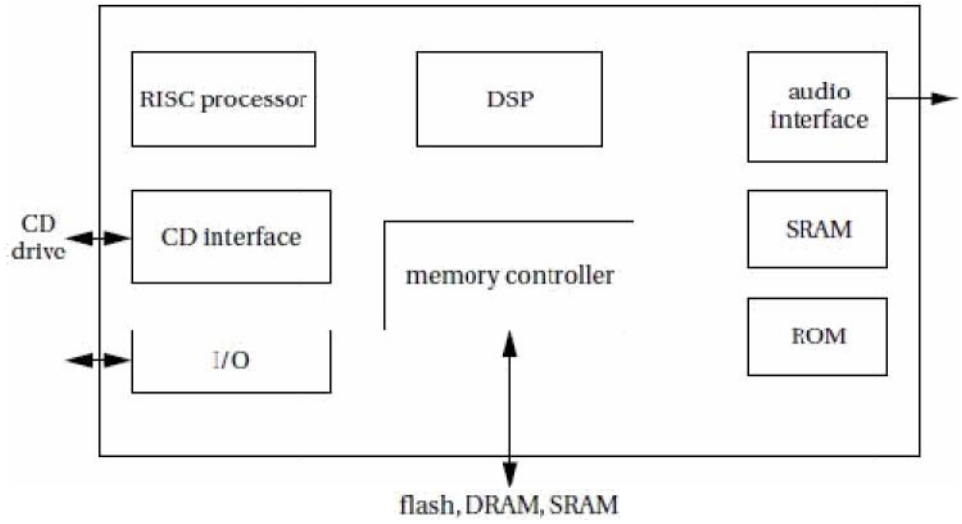


Fig 5.13 Architecture of Cirrcus audio processor for CD/MP3 players

The Cirrus CS7410 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.

5.4 SOFTWARE MODEM

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. As shown in Figure 5.14, 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.

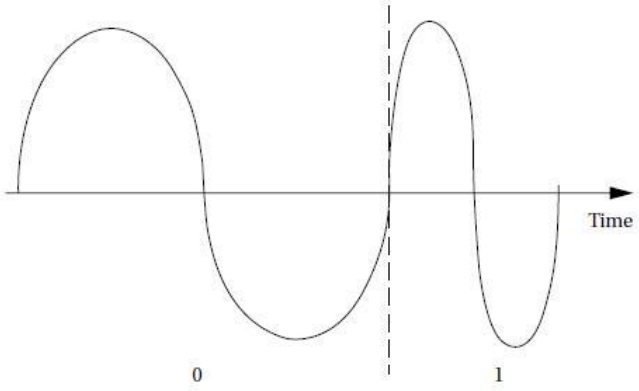


Fig 5.14 Frequency shift keying

The scheme used to translate the audio input into a bit stream is illustrated in Figure 5.15. 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.

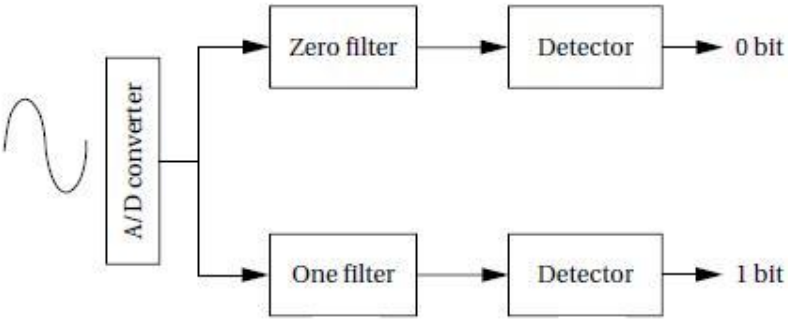


Fig 5.15 FSK Detection scheme

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.

The receiving process is illustrated in Figure 5.33. 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.

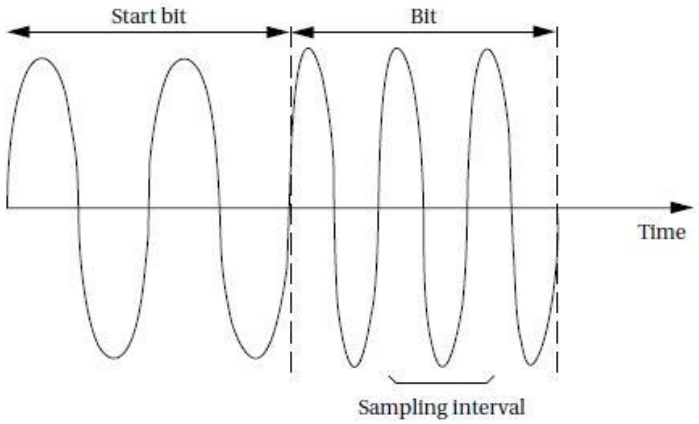


Fig 5.16 Receiving bits in the modem

Requirements:

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

Specification:

The basic classes for the modem as shown in the fig.5.17

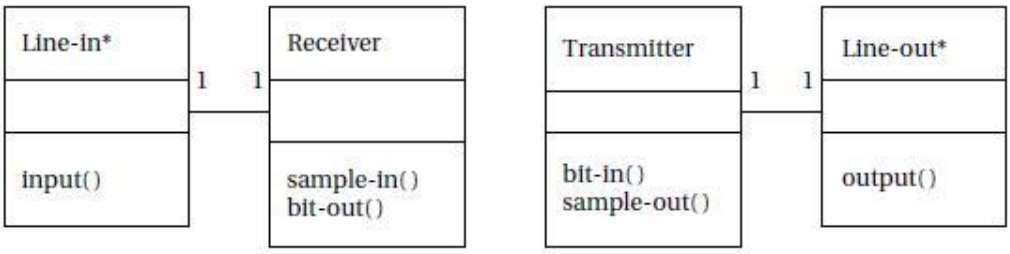
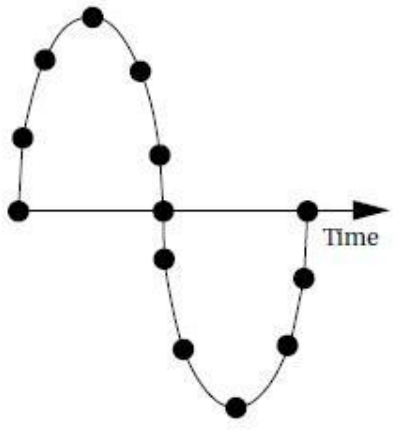


Fig 5.17 Class diagram for the modem

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 ***tablelookup***. Software oscillators can be used to generate periodic signals, but numerical problemslimit their accuracy. Figure 5.18 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.



Analog waveform and samples

```
float sine_wave[N_SAMP] =
    { 0.0, 0.5, 0.866, 1,
    0.866, 0.5, 0.0, –0.5,
    0.866, –1.0, –0.866, –0.5,
    0};
```

Table

Fig 5.18 Waveform generation by look up table

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.

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.

5.5 DIGITAL STILL CAMERAS

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.

Requirements:

Name	Digital still camera
Purpose	Digital still camera with JPEG compression
Inputs	Image sensor, shutter button
Outputs	Display, flash memory
Functions	Determine exposure and focus. capture image, Perform Bayer pattern interpolation, JPEG compression, store in flash file system.
Performance	Take one picture in 2 sec.
Manufacturing cost	Approximately $75
Power	Two AA batteries
Physical size and weight	Less than 4 ounces

System Architecture:

Typical hardware architecture for a digital still camera is shown in Figure 5.19. 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.

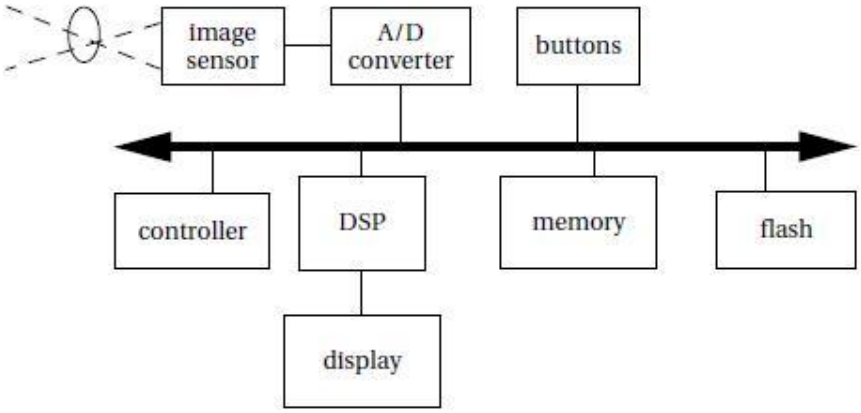


Fig 5.19 Architecture of digital camera

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 7.24 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.

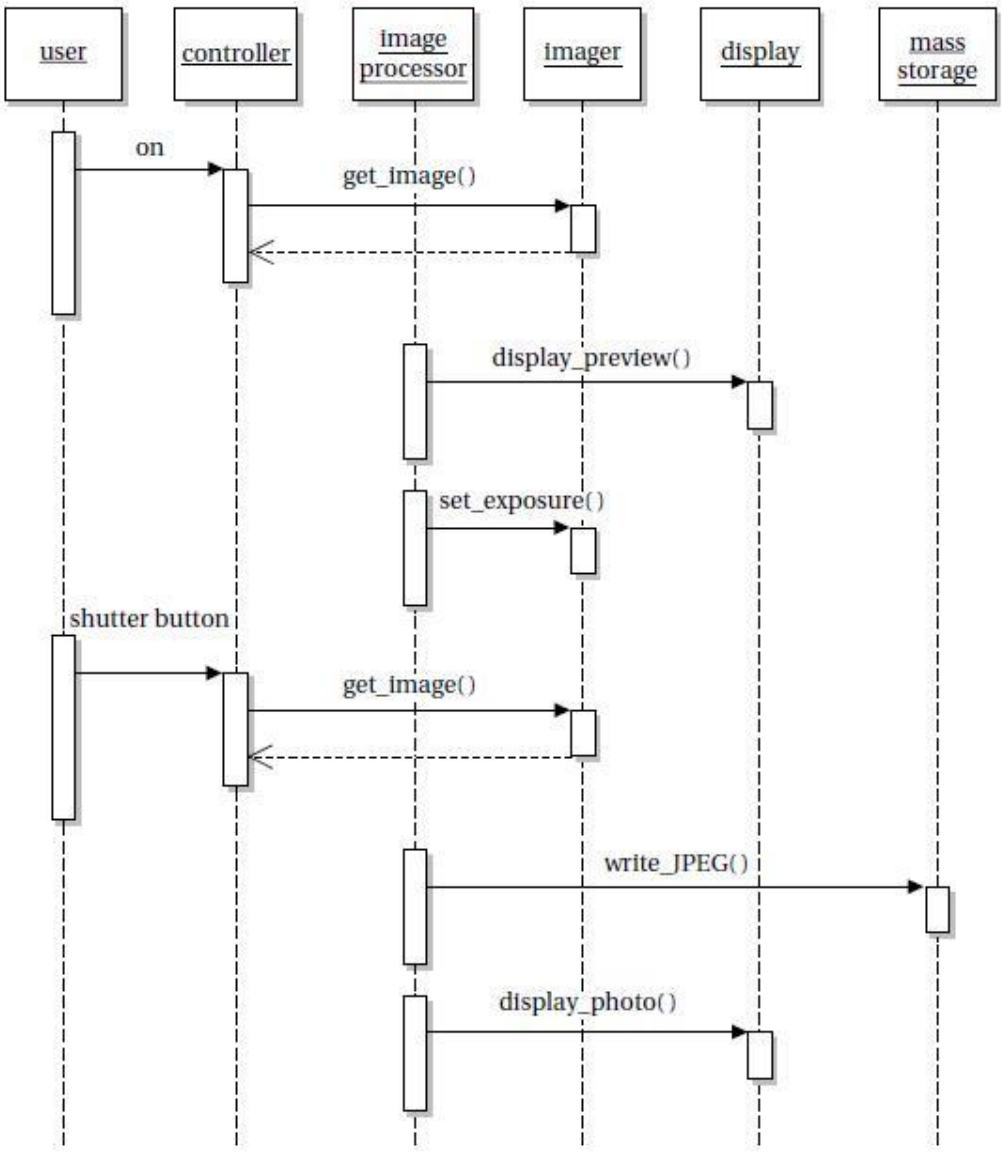


Fig 5.20 Sequence diagram for taking a picture with a digital still camera

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 5.21 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.

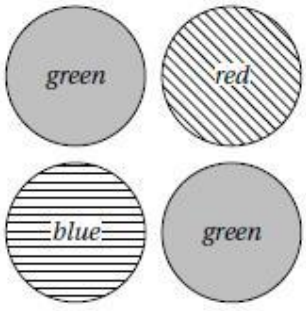


Fig 5.21 The Bayer pattern for color image pixel

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.

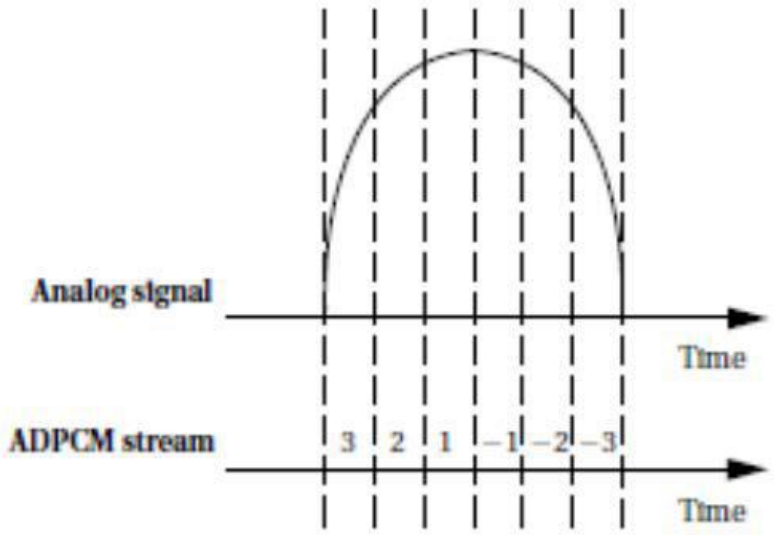
A buffer memory is used to capture the image from the sensor and store it until it can beprocessed by the DSP . The display is often connected to the DSP rather than the system bus.
Because the display is of lower resolution than the image sensor, the images from the image
sensor must be reduced in resolution. 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.

Telephone Answering Machine.

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



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 andthe actual signal value is minimized. AnADPCM compression system, including

anencoderand decoder, is shown in Figure. The encoder is more complex, but both theencoder 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; becausethe samples are differential, integration reconstructs the original signal. The encodercompares the incoming waveform to the predicted waveform (the waveform that will begenerated in the decoder). The quantizer encodes this difference as the best predictorof the next waveform value. The inverse quantizer allows us to map bit-level symbolsonto real numerical values; for example, the eight possible codes in a 3-bit code can bemapped onto floating-point numbers. The decoder simply uses an inverse quantizer andNintegrator 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 thesubscriber line is the **central office**. All information is carried on the phone line inanalog form over a pair of wires. In addition to analog/digital and digital/analogconverters to send and receive voice data, we need to sense two other characteristicsof 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 analogcircuitry to produce 0 for no ringing and 1 for ringing.

■Off-hook: The telephone industry term for answering a call is going **offhook**; thetechnical term for hanging up is going **on-hook**. (This creates some initial confusionsince off-hook means the telephone is active and on-hook means it is not in use, but theterminology starts to make senseafter 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.

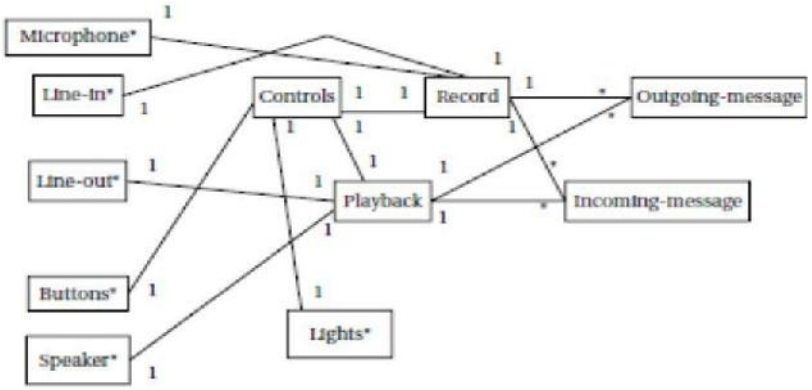
We can now write the requirements for the answering machine. We will assume that theinterface is not to the actual phone line but to some circuitry that provides voicesamples, off-hook commands, and so on. Such circuitry will let us test our system with a telephone line simulator and then build the analog circuitry necessary to connect to areal 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 thestart of every phone call.

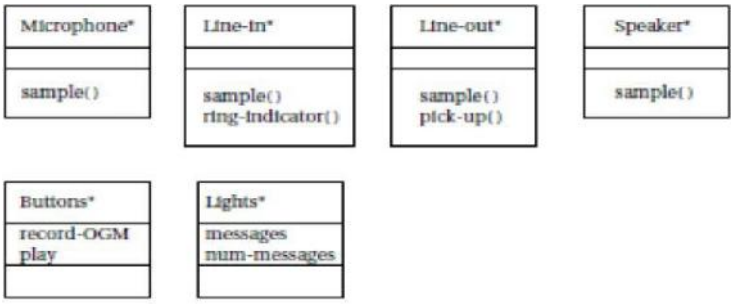
We have made a few arbitrary decisions about the user interface in these requirements.The amount of voice data that can be saved by the machine should in fact bedetermined by two factors: the price per unit of DRAM at the time at which the devicegoes into manufacturing (since the cost will almost certainly drop from the start ofdesign 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 muchsense to throw out old messages and replace them with new ones, and ideally the usercould select which protocol to use. Extra features such as an indicator showing thenumber of messages or a save messages feature would also be nice to have in a realconsumer product.

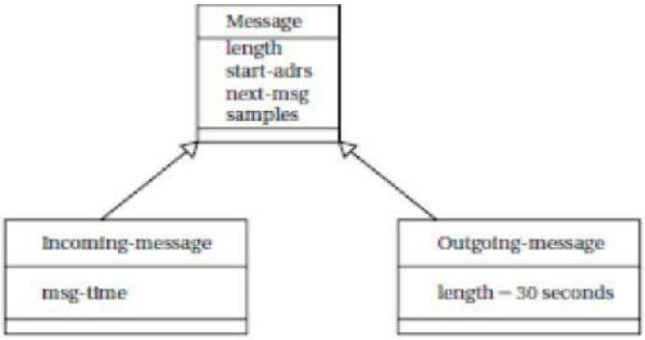
Name	Digital telephone answering machine
Purpose	Telephone answering machine with digital memory, using speech compression.
Inputs	*Telephone:* voice samples, ring indicator. *User interface:* microphone, play messages button, record OGM button.
Outputs	*Telephone:* voice samples, on-hook/off-hook command. *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.

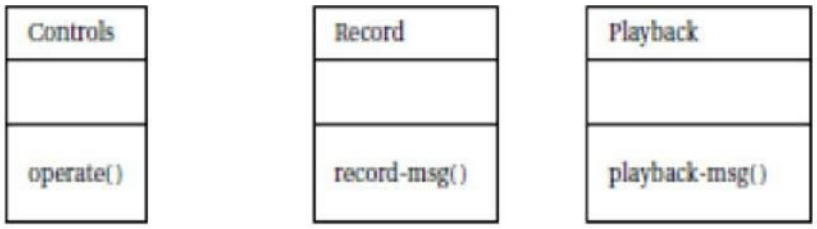
Specification:

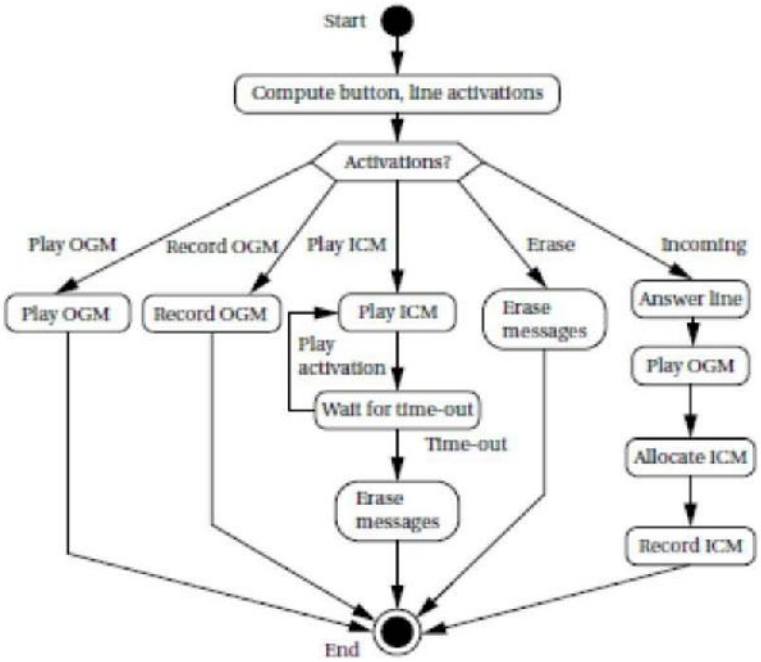
Figure1 shows the class diagram for the answering machine. In addition to the classes that perform the major functions, we also use classes to describe the incoming and OGMs. As seen below, these classes are related. The definitions of the physical interface classes are shown in Figure2. The buttons 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 thathandle 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 terminationcondition.


Microphone*
Line-In*
Line-out*
Buttons*
Speaker*
Controls
Record
Playback
Lights*
Outgoing-message
Incoming-message
1
*


Microphone*
sample()
Line-In*
sample()
ring-indicator()
Line-out*
sample()
pick-up()
Speaker*
sample()
Buttons*
record-OGM
play
Lights*
messages
num-messages


Message
length
start-adrs
next-msg
samples
Incoming-message
msg-time
Outgoing-message
length = 30 seconds



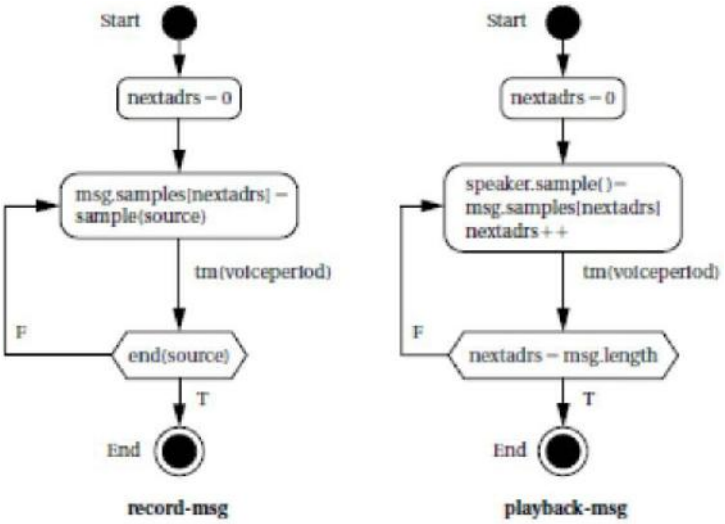


System Architecture:

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 andlights.
- The **speaker module** handles sending data to the user'sspeaker.
- The **telephone line module** handles off-hook detection and on-hookcommands.
- The **telephone input and output modules** handle receiving samples from and sending samples to the telephoneline.
- The **compression module** compresses data and stores it inmemory.
- The **decompression module** uncompresses data and sends it to the speakermodule.



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 mainloop.

```
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 interruptmodules.

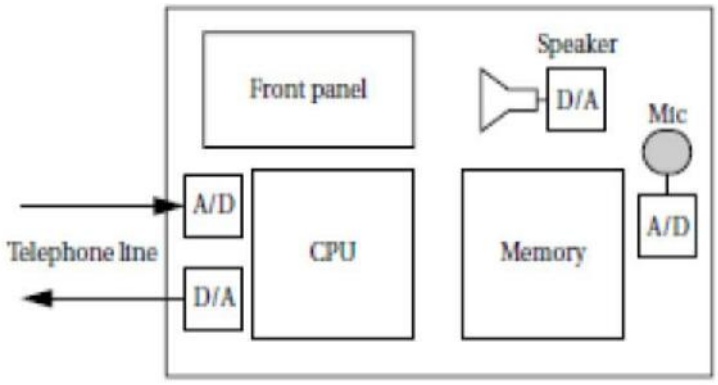
One subtlety is that we must construct a very simple file system for messages, since we have a variable number of messages of variable lengths. Since messages vary in length, we must record the length of each one. In this simple specification, because we always play back the messages in the order in 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 themessages.

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.

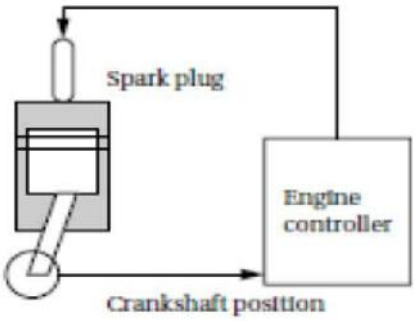
from a more fundamental message type. The major operational classes—Controls,

Record, and Playback—are defined in Figure4. The Controls class provides an operate

Automotive engine control:

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

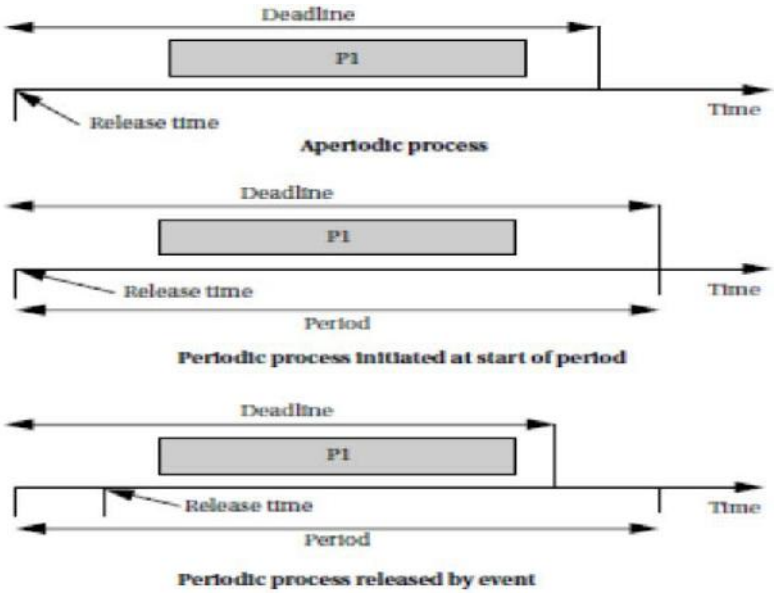
Timing Requirements on Processes:

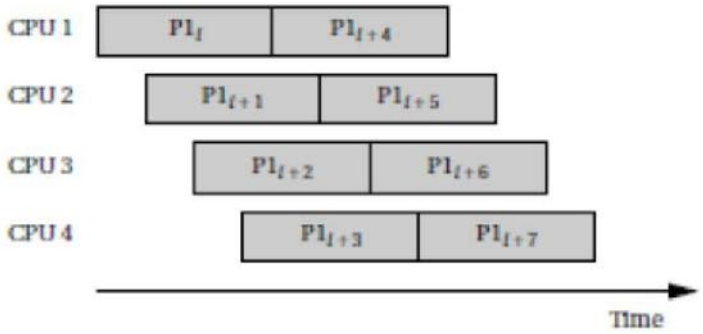
Processes can have several different types of timing requirements imposed on
Them by the application. The timing requirements on a set of processes strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether 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





1. 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

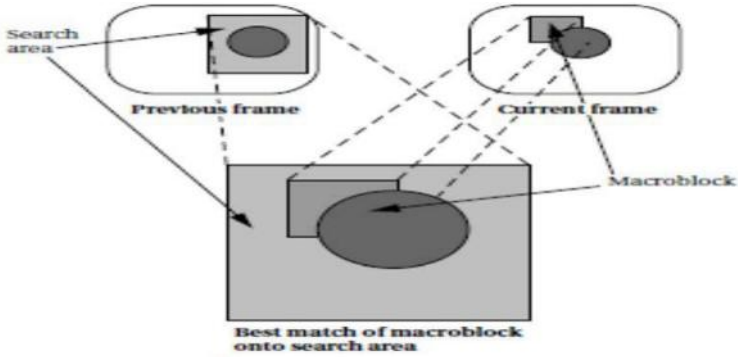
Algorithm and Requirements:

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, andox, 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.Wechoose 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;

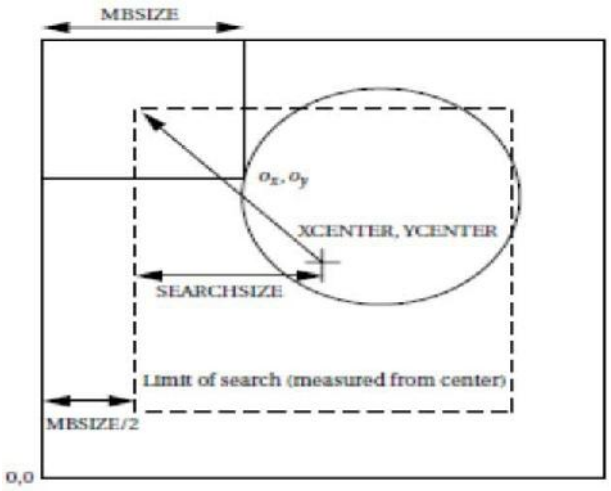
```
for (i = 0; i < MBSIZE; i++) { for (j = 0; j < MBSIZE; j++) {
result = result + iabs(mb[i][j] – search[i – ox
+ XCENTER][j – oy + YCENTER]);
}
}
if (result <= bestsad) { /* found better match */ bestsad = result;
bestx = ox; besty = oy;
}
}
```

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:

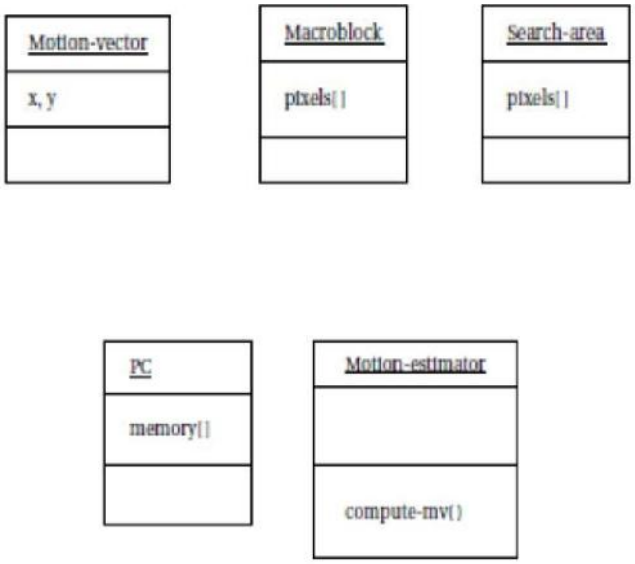
The specification for the system is relatively straightforward because the algorithm is simple. Figure defines some classes that describe basic data types in the system: the motion vector, the macroblock, and the search area. These definitions are straightforward. Because the behavior is simple, we need 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:

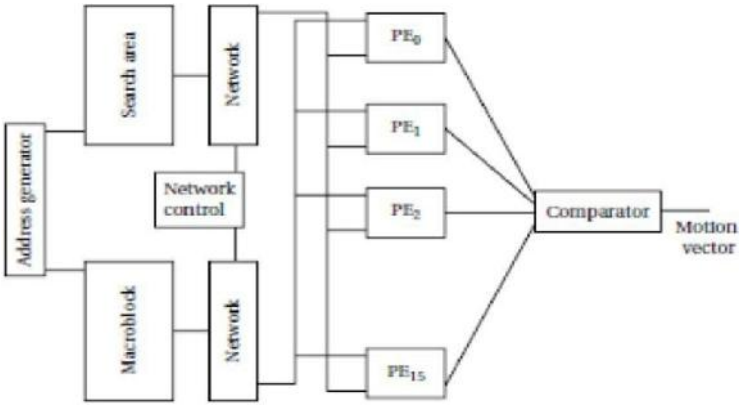
The accelerator will be implemented in an FPGA on a card connected to a PC's PCI slot. Such accelerators can be purchased or they can be designed from scratch. If you design such a card from scratch, you have to decide early on whether the card will be used only for this video accelerator or if it should 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 theschedule.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.

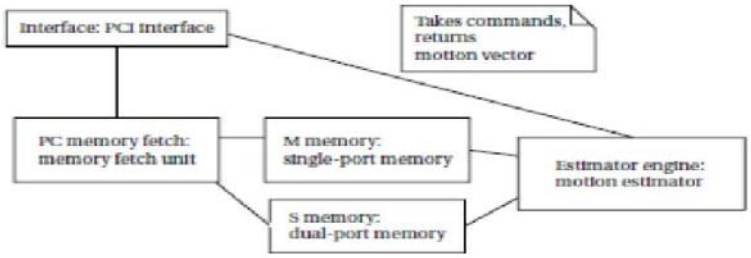


Component Design:

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.